DESIGN AND USE OF MANAGED OVERLAY NETWORKS

A Thesis
Presented to
The Academic Faculty

by

Sridhar Srinivasan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2007

DESIGN AND USE OF MANAGED OVERLAY NETWORKS

Approved by:

Dr. Ellen Zegura, Advisor
College of Computing
Georgia Institute of Technology

Dr. Nick Feamster
College of Computing
Georgia Institute of Technology

Dr. Mostafa Ammar
College of Computing
Georgia Institute of Technology

Dr. Mauricio Cortes
Member Technical Staff
Bell Labs

Dr. Constantine Dovrolis
College of Computing
Georgia Institute of Technology

Date Approved: 15 December 2006

To my family, for their support and encouragement.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and encouragement of people too numerous to enumerate, but I will try anyway. I would like to begin by expressing my gratitude to my advisor, Ellen Zegura, without whose patient guidance, probing questions and limitless support, I wouldn't have completed this journey. I would also like to thank Prof. Mostafa Ammar and Prof. Nick Feamster for their comments and insights. I would specially like to thank Prof. Constantine Dovrolis for his great lectures and the passion he brings to research. I would also like to thank my external committee member, Dr. Mauricio Cortes, for his guidance and for the opportunity to spend a very enjoyable summer in Bell Labs.

It is said that "many hands make lighter work" and my graduate studies were made much lighter because of the help and support of some wonderful colleagues. Special thanks goes to my collaborators, Richard Liston and Shashi Merugu, for their guidance and encouragement. I also thank Partha for his help in performing measurements till the very end. Beyond research, the people of NTG have also been instrumental in making my graduate life a whole lot of fun. For deep and wandering discussion on the meaning of life, I'd like to thank Abhishek, Amogh, Manish, Minaxi, Pradnya and Ruomei. Also, I'd like to blame my coffee addiction on Ravi and Amogh who collaborated to create the best coffee house in Atlanta in our lab. Srini, in between the time spent in helping everyone else in Atlanta, managed to find time to provide moral support, good food and Tamil movies to enliven GCATT, for which he has my gratitude.

The time I spent in Atlanta was enlivened by the great roommates that I had. Kapil showed me how anything could be burnt and yet taste good, while Biranchi

embodied the discipline that he imparted to us who sadly lacked it. Arumugam, who never met a sports statistic that he didn't remember, made all those football (and basketball and cricket and ...) sessions memorable while Karthik ensured that nothing went to my head.

None of these experiences would have occured if not for the generous support of my parents and my brothers and sister. They imparted to me a sense of wonder about nature and encouraged my curiosity and imagination and gave me the freedom to pursue it as long as I wished. For all this and more, I do not have enough words to thank them with.

Lastly but most importantly, I'd like to thank my wife, Jayashree, for standing by me. Her patience and encouragement have made this thesis possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

As the role of the Internet has been steadily gaining in importance, overlays are increasingly being used to provide new services and to deploy older ones. Some of the services for which overlays have been proposed include multicast, quality of service (QoS), search, and resilient networks. The use of overlays, in turn, has led to more interest in improving their performance. The performance of an overlay network depends significantly on how the network is structured, i.e., the placement of the nodes in the underlying network topology, the links between the overlay nodes and the access links of these nodes. This thesis focuses on algorithms for improving the performance of *managed overlays*, in which the capabilities and locations of the participating nodes are known. Managed overlay networks allow for greater optimization and design than unmanaged overlays due to this increased knowledge of the network.

Our first technique applies to managed overlays that are used for streaming real-time content. Many such networks are composed of end-systems with a single access link to the rest of the network. We propose Time Division Streaming (TDS) as a technique to schedule the serial access to these links. We show that overlays constructed using TDS reduce the average delay experienced by the participating nodes as compared to previous designs.

Our second technique focuses on the placement of nodes in service overlay networks. We propose RouteSeer, an algorithm to place nodes in service overlays to create resilient paths between overlay nodes. RouteSeer works by examining local routing information at the overlay nodes to place a few intermediate nodes such that the overlay links are protected by a disjoint path through an intermediate node. We

extend RouteSeer to topologies with incomplete and dynamic routing information and demonstrate its ability to protect overlay links in such cases as well.

Finally, we propose and evaluate a managed overlay network for a highly reliable, multi-attribute query service. As opposed to previous designs, our architecture ensures bounded query response times and can handle dynamically updated data.

CHAPTER I

INTRODUCTION

The role of the Internet has been steadily increasing in importance as more and more applications and services are implemented over it. Perversely, the growth of the Internet and its importance have made it harder to make any changes to the underlying protocols and architecture of the Internet to provide new services or deploy existing services. *Overlays* have emerged as the preferred way to work around this stasis by providing new services such as resilient networks [6], rendezvous [75], and search [26, 38] on the Internet. Overlays have also been used to deploy older services such as multicast [20], and QoS [77, 81] that have proven difficult to provide in the IP network layer.

An overlay network comprises *overlay nodes* that are responsible for routing and forwarding, connected by *overlay links* that correspond to paths in the underlying network. An illustration of an overlay network is shown in Figure 1 in which the labeled nodes represent overlay nodes. A link between overlay nodes consist of a path in the underlying network, shown by the sequence of solid edges.

Overlay networks can be quite diverse with widely differing performance criteria.

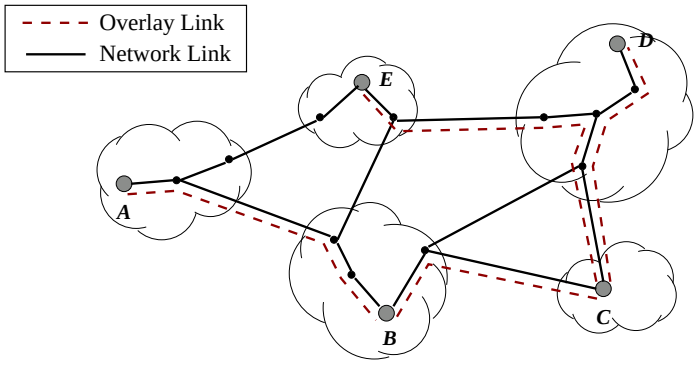


Figure 1: Example of an Overlay Network

An example of an overlay network is the Gnutella [26] file sharing network, in which the overlay nodes are the end systems. These nodes connect to each other to form an overlay network on which queries from nodes are forwarded by flooding. Other examples include research networks such as PlanetLab [58] and commercial networks such as Akamai [2]. The PlanetLab network consists of approximately four hundred nodes and allow users to configure the nodes into overlays for specific applications. The Akamai network of content distribution servers can also be considered an overlay network composed of approximately 15,000 nodes dedicated to distributing content to end users from nodes nearest to them in the network topology. The SureRoute service [3] which runs on the Akamai network provides resiliency to websites by routing their traffic through an intermediate Akamai node when the direct path to the website is down.

1.1 Classification of Overlay Networks

Overlays can be classified in several ways. One way to perform the classification is on the type of nodes that comprise the network as shown by the rows of Table 1.[1] In general, overlay networks, such as Gnutella, in which the nodes are end-hosts are called *peer-to-peer* networks. These nodes are connected to the overlay network for highly variable, user-defined times. The nodes form the clients of the overlay service as well as the routing and forwarding infrastructure. In contrast, in *service* or *infrastructure* overlays such as Akamai, the overlay nodes are distinct from the clients of the service. These dedicated nodes are usually connected to the overlay for long periods of time and are typically managed by a single entity. The use of dedicated nodes can be exploited in the algorithms used to design service overlays unlike peer-to-peer algorithms which have to assume that the nodes are likely to fail.

We can also classify overlay networks on the knowledge that each node in the

[1] We discuss the different classifications of overlay networks in detail in the next chapter.

Table 1: Classification of Overlay Networks

	Unmanaged/Decentralized	Managed/Centralized
End hosts	Peer-To-Peer networks, e.g., Gnutella	Centralized networks, e.g., Narada
Dedicated hosts		Service Overlays, e.g., Akamai

overlay network has of the other nodes in the network as shown by the columns of Table 1. In some networks, which we call *managed networks*, all the nodes, their capabilities and, in some cases, their locations, are known to each of the nodes in the network. On the other hand, in unmanaged overlays, no node has knowledge of all the nodes in the network. Managed overlays, especially if composed of dedicated nodes, offer more scope for node placement and network design mainly because of the availability of this global knowledge, either in a centralized or distributed fashion. Unmanaged overlays tend to be constructed with nodes using only local knowledge to connect to the network.

The performance of an overlay network, either managed or unmanaged, depends significantly on how the network is structured.[2] Some of the factors that affect the performance of an overlay are:

- **Placement of nodes:** The performance of the overlay can be significantly affected by the location of the overlay nodes in the underlying network. It is also affected by the placement of the overlay nodes relative to each other and to the clients, if they are different from the overlay nodes. Placement affects the performance of overlay links between nodes in terms of the quality and diversity of the underlying network paths and the load on these paths.

- **Links between nodes:** The set of overlay links that are maintained in the overlay network also affects the performance of the network. In the case of small

[2]We will use the term "performance of the overlay network" to informally mean the performance of the overlay on a metric of interest to the applications using the overlay, such as latency, resiliency of paths and so on. The measure of performance will be defined later for each overlay we discuss.

overlay networks, it is possible for each node to know about all other nodes, in which case the overlay network may be fully connected. In large networks with thousands of nodes, it is often not possible for each node to keep track of all other nodes in the system, in which case the creation or selection of links between overlay nodes is an important problem.

- **Access links of nodes:** The network access links of the nodes in overlay networks can be diverse, ranging from hosts connected to the Internet through gigabit links to hosts using a cable modem or DSL link. This network link is usually the bottleneck for transferring data when the node is part of an overlay and becomes a resource that must be shared between the various overlay links that use it, especially when the application requires large bandwidth such as multi-player games, conferencing and file or content distribution. Other factors such as the processing power of the nodes and the load on the overlay nodes also affect the performance but typically, the access links of the nodes are the limiting factor of the performance of the nodes.

Out of the three factors outlined above, the placement and access links of nodes have received little attention in research. These factors are especially important for managed overlays. This thesis is an attempt to remedy this by concentrating on the design opportunities offered by managed overlay networks.

1.2 Contributions

This thesis focuses on techniques for improving the performance of *managed overlays* which include service overlays and centralized overlays, i.e., networks in which all nodes have knowledge of the entire overlay. Global knowledge of the nodes can be used to design overlays based on node capabilities while managed overlays with dedicated nodes offer the possibility of network design by carefully placing nodes in the underlying network. In this dissertation, we propose two techniques for improving

performance in managed overlays.

1.2.1 Time Division Streaming

Our first technique can be applied to overlays used for streaming real-time content similar to the End System Multicast overlay called Narada [20]. These overlays are composed of end systems which can have varying access links varying from a DSL line to gigabit links. Previous research has mostly ignored the effect of serial access to this link and its effects on overlay construction and use. We show that the effect of this serialization can be significant when constructing overlays for delay-sensitive data.

We propose a technique called *Time Division Streaming* (TDS) to optimize the use of this access link by scheduling the data traversing this link among the competing overlay links. TDS allows each overlay link to, in turn, send large blocks of data through the access link. We show that overlays constructed using TDS can substantially reduce the average delay experienced in receiving data by the nodes in the overlay.

1.2.2 RouteSeer

Our second technique applies to service overlays composed of dedicated nodes. Much of the current research in designing overlays assumes that the nodes that comprise the network already exist on the network in some form and do not go into the question of selecting the nodes (more precisely, their location in the network topology). Most research focuses on constructing the links between the overlay nodes based on performance metrics such as delay, loss or resiliency.

We propose a two-part technique for placing overlay nodes in service overlays with the objective of ensuring that each path between pairs of overlay nodes is "protected" by an indirect path using an intermediate node. Having such disjoint paths has been shown to improve performance of the overlay [7, 25]. In our method, overlay nodes

are first placed "close" to the clients of the service overlay. We then use a technique called *RouteSeer* to place intermediate nodes by examining the routing tables at the overlay nodes such that most overlay links are protected by a disjoint path through an intermediate node. Using simulations, we show that RouteSeer can substantially increasse the protection of overlay links.

1.2.3 Path Diversity in Networks

Using a network model developed for RouteSeer, we examine the general problem of path diversity of which overlay resiliency is a component. We examine other approaches to obtaining path diversity, including multihoming [4] and proposals for multi-path routing using BGP [86]. We show that overlay networks allow for the most flexibility in designing protected paths.

We evaluate the performance of these path protection schemes using network characteristics such as packet loss and latency. For this, we evaluate managed overlay networks designed with RouteSeer using incomplete information from routing tables as well several multihomed sites. We show that the managed overlay networks can perform well in recovering from path failures without being penalized on their latency performance.

1.2.4 LUNA: Design of a managed overlay

We use some of the ideas outlined above to design a managed overlay to provide a resource location service that maps resource names to network locations. The service, called LUNA (for LookUp Network Addresses), can be used by applications such as VoIP which need to store and retrieve highly dynamic user data using queries on attributes such as network location, geographic coordinates, allowed features, etc. LUNA is a highly scalable and reliable service which allows for searching on multiple attributes and provides bounded response times for queries. We propose an architecture for LUNA based on using a single overlay hop to reach the node storing the

record. We demonstrate the advantages of our design for dynamic data and compare our design for this service with other approaches.

1.2.5 Beyond Managed Networks

Though we propose TDS and RouteSeer in the context of managed overlays, they can be applied to unmanaged networks such as unstructured peer-to-peer networks as well. For example, nodes in peer-to-peer networks are generally end systems with restricted access links capacities. When an application such as media streaming is using such a network, the opportunity for scheduling this data on the access link can be exploited by TDS. Also, the large number of nodes in such networks creates a diversity of overlay paths, which presents its own difficulty in that the information about alternate paths to be maintained can be very large [25, 32], either in terms of the cost of probing different paths or routing state to be maintained. RouteSeer can be used to reduce the amount of state that needs to be maintained at the nodes.

We will highlight the applicability of these techniques to other networks as we discuss them in subsequent chapters.

1.3 Thesis Organization

We next present some background on overlay networks and survey the work related to the design and use of managed overlay networks. Chapter 3 presents the Time Division Streaming technique and its evaluation. We present RouteSeer and discuss its effectiveness in providing overlay path protection in Chapter 4. We then proceed to examine the various path diversity mechanisms in Chapter 5 and also present an evaluation of user-perceived performance.

Chapter 6 presents a specific managed overlay network called LUNA that we design for fast and efficient mapping of resource names to network locations. We summarize the work in the thesis in Chapter 7.

CHAPTER II

OVERLAY NETWORKS

Overlay networks have been around in various forms almost since the beginning of the Internet. One of the earliest examples of an overlay network was a proposal to run OSI protocols between sites using the IP network as the subnetwork [34]. There has been a tremendous amount of work in the past eight years on various designs and uses of overlay networks. In this chapter, we present a brief survey of overlay networks, concentrating on various designs for overlay networks and some of the applications for which these networks have been proposed.

2.1 Overlay Classification

Classification of overlay networks is a hard task as there are a large number of proposals and working overlay networks, each designed for a specific objective with a different set of assumptions about its constituent nodes and links. Previously, the main classification of overlay networks has been based on the structure of the network. With the increase in overlays with dedicated nodes, that single axis of classification is insufficient to capture the variety of overlay networks that exist.

To understand the design of overlay networks, we classify overlays along three different axes, each capturing a different aspect of a network's structure. The axes are based on the type of nodes, the structure, management and design of the overlay networks. For each axis, we discuss some of the features and assumptions of the overlay networks that lie on that axis. Note that these axes are somewhat orthogonal, and so the same networks can appear in different classifications.

2.1.1 Type of nodes

Overlays are characterized by the nodes that compose them. In the last century, most overlays were composed of nodes dedicated to the overlay. With the recent increases in conectivity to the Internet by end hosts, many overlays are composed only of end systems managed by users. These peer-to-peer networks have seen wide acceptance and use.

The first overlays were composed of nodes that were usually dedicated to the functioning of the overlay. More recently, interest in such networks has increased for providing new and improved service on the Internet. These nodes tended to be connected to the overlay for long periods of time and had low failure rates. Thus, the algorithms for these overlay networks could mostly ignore node behavior. The nodes in these networks also tend to have better connectivity to the Internet. Some overlay networks that fall in this category include ALMI [69] and OMNI [9].

Peer-to-peer networks are composed of user-controlled end hosts. Therefore, their lifetimes on the peer-to-peer network vary and these networks have high churn in the nodes that constitute them. Examples of such networks include Gnutella [26], Chord [76] and SplitStream [17].

To reduce the impact of the high churn, some peer-to-peer networks propose a two-level architecture in which a few dedicated nodes, which connect to the network for longer periods of time, are used to improve the performance of the network, e.g., supernodes in KaZaa [38].

2.1.2 Structured vs. Unstructured

The organization of overlay networks can vary widely. In small networks, it is possible to have the nodes keep track of all the nodes in the overlay. This approach quickly becomes impossible as the size of the network increases. For large networks (nodes numbering greater than a few hundred), two different strategies are used:

unstructured and structured.

In unstructured networks, as the name suggests, the network has no defined structure. When nodes join the network, usually be connecting to another node already on the network, they connect to a set of neighbors which they discover by querying or by monitoring traffic. Nodes continually adjust the set of neighbors based on various performance criteria.

Operations on these networks are performed by "flooding" the request through the network. A node forwards the request to all its neighbors except the one from which the request came in. The request is propagated for a specific number of hops to limit resource usage. Gnutella [26] and KaZaa [38] are famous examples of such networks.

Structured networks have a rigid set of rules about the nature and number of neighbors for any node. Each node is assigned a unique key and based on this key, connects to the network topology at a specified location. The location also dictates the neighbors for the node based on their keys. These keys are usually generated using a uniform hash function on the node location or record identifier. These networks are also called Distributed Hash Tables (DHTs) [76, 41, 64, 63].

When a node searches for a key, it forwards the query to the neighbor whose key is most similar to the query. This continues until the query reaches the node responsible for the searches key, which either responds with the record or returns an error. This search process is very efficient and the number of hops required generally corresponds to some function of the network size. More recently, there have been proposals that reach the destination in one or two hops [33], irrespective of the size of the network.

Before these distributed structured network designs were proposed, a centralized structured network called Napster [54] was deployed. In this design, each node in the overlay would register its location and the content stored on it with a centralized server. All queries were sent directly to the central server which replied with a set

of nodes that stored the keys. The querying node would contact nodes from this set to obtain the content. This design has many limitations including the dependence on a single central entity, which were resolved by structured overlay networks. It can also be argued that this design has more in common with a client-server architecture, especially in the query mechanism.

2.1.2.1 Hybrid Architectures

Structured and unstructured networks are two ends of a spectrum of network organization. Structured networks offer efficient search mechanisms while unstructured networks offer simplicity and robustness to node and network failures. To bridge this gap, several proposals to improve the performance of unstructured networks have been put forward that have optimizations to propagate searches more efficiently.

These proposals include mechanisms such as creating small world graphs in which nodes connect mostly to other close nodes but also to a few random nodes [57, 51]. Other proposals [19] use properties of power-law graphs to achieve good performance.

2.1.3 Managed and Unmanaged Overlays

Overlays can also be classified based on the control and management of the nodes in the overlay.

In some overlay networks such as ALMI, the design and control of the network is centralized with a single node managing the links and data on all the nodes in the network. In others such as Narada [20], all nodes have complete kowledge of the network topology and can make decisions on links based on this global knowledge. This knowledge and control of the network allows for global performance optimization and the use of centralized algorithms for the design of the networks. We call such networks managed overlays. Managed overlays tend be relatively small with all nodes having knowledge of the entire overlay, or all nodes are controlled by a single overall entity, e.g., Akamai.

In unmanaged overlays, the management of the links and selection of neighbors is localized to the individual nodes, allowing only for local optimizations. Most peer-to-peer networks fall into this unmanaged overlays category.

2.1.3.1 Designed vs. Oblivious Overlays

We can further classify managed overlays based on the extent of design in construction of the overlays. The design of overlay networks incorporates both node placement and topology design. Most overlays are oblivious, i.e., the overlay is created using nodes already present in the underlying network. These overlays are restricted to selecting the nodes that form the overlay topology and to manage the links among the nodes.

Recent proposals [35, 18] for placing the nodes that constitute the overlay have allowed for doing both node selection and topology design. We discuss these proposals in detail in the next section.

2.2 Related Work

In this section, we discuss some overlay designs and protocols related to this thesis in greater detail.

2.2.1 Overlay Design

One objective in overlay design is to provide resilient paths between the overlay nodes. This can be used as a basis for further optimizations of other time-dependent metrics such bandwidth and loss. Work in this area can be split into algorithms for oblivious networks and designed networks.

2.2.1.1 Node selection

The approaches outlined in this section focus on *reactively selecting* an intermediate node to provide an alternate path between two end points rather than *proactively placing* intermediate nodes. These algorithms are for node selection rather than

node placement as they were designed for peer-to-peer networks with a large set of intermediate nodes to choose from.

Nakao et al. [56] proposed a routing underlay that could be queried for topology information by multiple overlays. One of the routing services that the underlay supports is to find disjoint paths between nodes. Their approach uses topology information gathered from BGP tables at the nodes to create a graph of the network. This graph is used to compute edge disjoint shortest paths through the intermediate nodes. Topology information from the intermediate nodes is used to verify the paths are disjoint.

In a different approach, Gummadi et al. [32] propose a *random-k* policy in which k intermediate nodes are selected at random and in case the direct path from a source to a destination fails, these intermediate nodes are used to forward the data. They showed that for $k = 4$, it is possible to recover from most failures of the direct path. They compare this policy with one that orders the intermediate nodes based on the number of common ASes called BGP-k, which assumes that this path information is available, and show that random-k is close to BGP-k. This is achieved without assuming knowledge of the network topology.

Fei et al. [25] propose a different policy to select intermediate nodes for disjoint paths by using the AS-level path information of the source-destination and source-intermediate node paths. Their approach is useful in cases when the intermediate node is selected before or during connection initiation. Their heuristic, called *earliest divergence*, uses this path information to select the intermediate nodes which diverge earliest from the source-destination path. They compare their approach to the random-k policy and show that a variant of the earliest divergence heuristic, which narrows the set of intermediate nodes using earliest divergence and then chooses one randomly from the remaining, reduces the overlap the most.

2.2.1.2 Node Placement

There has been some work on placing nodes for designed networks beginning with Han et al. in [35]. They performed a measurement based study into the question of how to place overlay nodes to obtain overlay network resiliency in the face of network layer disruptions such as network node or link failures. The authors use measurements to answer the questions: how many different ISPs should have overlay nodes in them and in each ISP how many overlay nodes should be placed to get path deiversity. The measurements for this technique require a substantial overhead in active probes and also requires access to all the possible intermediate overlay node locations. It is also not clear how the initial set of intermediate overlay locations can be selected.

Another closely related work [18] also examines the same problem using a graph-theoretic approach. The problem the authors consider is that of placing relay nodes inside ISPs to ensure path diversity between intra-AS origin-destination pairs. They define a measure of penalty between a direct path from source to destination and an indirect path through a relay node to indicate the number of links that are present in both paths. Their objective is to reduce the overlap with the indirect paths over all source-destination pairs. For this, they propose two heuristics based on incrementally adding nodes to the set of relay nodes. This work assumes that the network topology is available to compute the penalty for the potential relay nodes and is restricted to intra-AS paths and leaves open the question of placement for inter-AS paths.

2.2.2 Applications on overlay networks

Overlay networks have been proposed for many applications, including search, resilient networks, and multicast. In this section, we will concentrate on two applications, overlay multicast and search. We will leave the discussion of related work in resilient networks and routing to Chapter 4 at which point we would have more context to discuss these applications.

2.2.2.1 Overlay Multicast

Overlay multicast has generated a lot of interest because of the desirability of multicast and the lack of widely deployed multicast networks. There are many proposals for deploying multicast using overlay networks in both managed [20] and unmanaged [8, 36, 59] forms, but in this section, we will restrict the discussion to managed overlays.

In [69], the authors describe the problem of creating minimum diameter degree bounded spanning trees and show that it is NP-Hard. They propose a greedy heuristic to create trees based on this objective. In [45], the authors define the cost of a tree as the number of special proxy nodes used to create multicast trees. Using this they propose to create trees which satisfy a maximum delay bound while minimizing cost. They provide an optimal solution for graphs with uniform edges and show that this problem is NP-Hard in the general case with non-uniform edges.

In [9], the minimum average-latency degree-bounded directed spanning tree problem is introduced in context of a two-tier infrastructure for implementing large-scale media-streaming applications. The infrastructure, called OMNI (Overlay Multicast Network Infrastructure) consists of a set of Multicast Service Nodes (MSNs) to which end-hosts connect to form the multicast tree. The objective of this work is to reduce the average latency to the end-hosts. This is achieved by arranging the MSNs to create minimum latency trees where each MSN is weighted by the number of clients connected to it. The authors impose a degree bound on each MSN but do not account for the transmission delays at the MSNs which we consider in this work. Also, we focus on application-layer multicast trees without any explicit infrastructure in the network.

In [15], the authors point out that the models used currently to construct these trees neglect to consider the fact that the most nodes in an end-system multicast tree have a single network connection and this connection has to be shared between all the

children of the node. The authors propose an overlay network model to account for these costs and propose heuristic algorithms to construct multicast trees that consider the transmission and computation delays at each of the nodes in the multicast tree. The overlay model proposed does not explicitly consider the degree constraints at nodes. The construction of the tree is based on minimizing the delay to hosts but does not consider the effect of the degree constraints imposed by the access link bandwidth of the nodes or the effect of the access link scheduling on the average delay of the nodes in the tree.

2.2.2.2 Search and lookup

One of the primary applications for overlay networks has been resource location, more specifically searching for or looking up records. The Gnutella and Skype networks can be considered to primarily be resource location services where the resources happen to be files and users, respectively. Though these networks are popular, they suffer from several shortcomings. Since queries are flooded, in case the record being searched for is beyond the horizon of the query, the querying node will not get a response. There is no bounded time in which a response or error is generated, making these networks unsuitable for time-dependent applications.

There have been proposals for other unstructured peer-to-peer systems that aim to remedy some of the flaws of these systems, specifically to improve the routing of queries. These proposals (for example, [57, 42, 30, 44]) and others such as Gia [19] propose mechanisms to improve the efficiency of queries. Though these designs provide improvements over unstructured networks, their query routing still requires several overlay hops to reach their destinations.

Gupta et al [33] propose a one-hop structured overlay such that any query can be answered by taking a single hop to the node storing the record for the identifier. In this one-hop overlay, each node maintains a routing table with information about

all other nodes in the network including the address space that it is responsible for, allowing it to get to the destination in a single hop. To maintain this routing table, changes in the set of nodes are propagated using a hierarchy imposed on the nodes. The address space is divided into slices with a node in each slice being designated as slice leader. Changes in the set of nodes (joins and leaves) are sent to the slice leader, which then disseminates the changes to all the other slice leaders who, in turn, send these overlay update messages to the nodes in their slice. The one-hop overlay is designed for a dynamic environment with many and frequent node changes and uses hashed identifiers for its records.

There are other proposals related to resource location. The DNS hierarchy can be considered to be one such mechanism. Ramasubramanian and Sirer [62] proposed CoDoNS as a replacement for DNS to provide low latency query and fast update performance. The CoDoNS system uses a combination of a Pastry-based DHT and proactive caching of records to deliver this performance. All record identifiers are hashed and the hashed identifiers are inserted into a Pastry DHT. The Pastry DHT resolves queries by forwarding each query to the node that matches the identifier's prefix with one more digit that the current node. CoDoNS uses this routing procedure to identify the caches where a particular identifier is to be cached. Based on the query characteristics of the identifiers, the more popular identifiers are cached at all nodes that match shorter prefixes of the identifier, with the most popular identifiers cached at almost all the nodes. This is shown to work well with Zipf-based distributions such as the DNS query patterns. Updates in CoDoNS are propagated in a similar fashion, thus, the time to update the most popular query would be the time to take $\log N$ hops in the Pastry network, where N is the number of nodes in the network.

The search mechanism does not have to restricted to a single key. In Mercury [13] , the authors propose a multi-attribute overlay using one overlay for each attribute to answer queries on multiple attibutes. Each node in Mercury is a part of several

overlays and each overlay is organized into circular ring based on the identifier space (without any randomizing hash functions). Routing of messages on each overlay is based on a small-world network and takes approximately $O(log^2 n)$ hops where n is the number of nodes in the overlay.

2.2.3 Objectives of this dissertation

The objectives of this dissertation are to propose new algorithms for the design of managed overlay networks. The algorithms are proposed in the context of different overlay applications.

For the work in Chapter 3 to optimize the performance of overlay multicast in managed overlays, we use the model developed by the authors in [15] as a starting point for our algorithms to schedule the use of the access links as discussed in Chapter 1.

The existing work in overlay design shows that node placement for path diversity in overlay networks remains unsolved. In our work in Chapter 4, we propose and evaluate a new algorithm for node placement while outlining some related work on routing and path diversity. We explore its implications on network path diversity in Chapter 5.

In Chapter 6, we design a managed overlay to improve search and lookup for extremely large user populations. Our design overcomes some of the shortcomings outlined in Section 2.2.2.2.

CHAPTER III

TIME DIVISION STREAMING

3.1 Introduction

The increase in video content on the Internet has led to the development of various specialized networks for the distribution of video content such as content distribution networks [2]. Multicast is a natural method to distribute such content without dedicated infrastructure. Unfortunately, wide-area multicast is not widely deployed and so application-layer multicast [20, 60, 8, 59, 17] has been proposed as a viable alternative for deploying large-scale multicast. In application-layer multicast, a set of nodes collaborate to form an overlay over which the content is distributed. The end hosts that compose the overlay network are responsible for creating and maintaining the multicast tree and also forwarding the data to their children in the tree. The application-layer multicast proposals include both managed and unmanaged networks. Applications that benefit from the use of application-layer multicast include media streaming, multi-player games, conferencing and file or content distribution.

Important metrics for these applications are the delay and jitter experienced during data transfer. In the case of file distribution applications, the average time to obtain the file is also an important requirement. For this reason most application-layer multicast schemes concentrate on creating multicast trees with low latency paths.

The end hosts that form the multicast tree are often connected to the rest of the Internet using a single access link such as a DSL or cable modem line [74]. The single access link is a shared resource that must be scheduled among the children of the node. This scheduling can affect the time taken to transfer data to the child nodes. As a simple example, consider a case in which a source node r with a single access

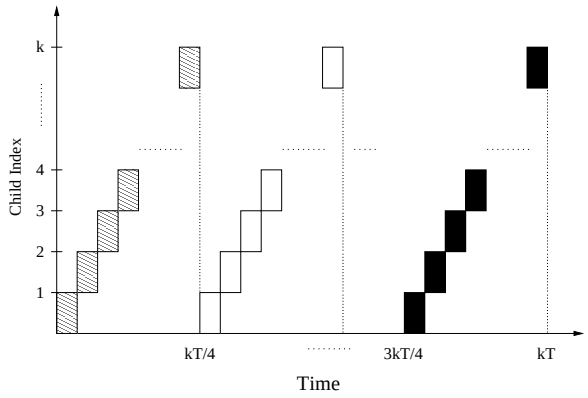


Figure 2: A block comprising of four packets being sent to k children a packet at a time

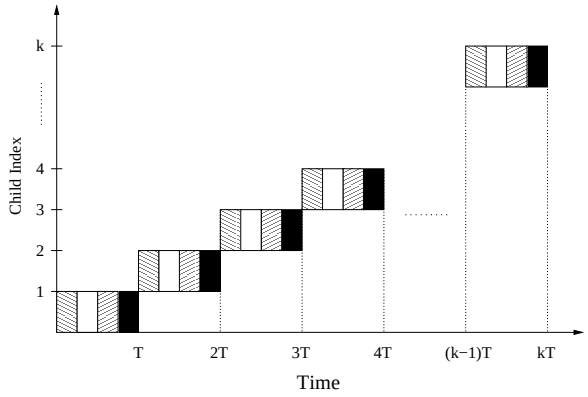


Figure 3: A block comprising of four packets being sent to k children a block at a time

link to the rest of the network, transfers a block of data to its k children. Let us further assume that this block is composed of several packets. Consider two different simple means of scheduling access to the link: in the first scheme, the block is sent one packet at a time to each child in turn and in the second, the entire block is sent to a single child at a time.

The methods of delivery are illustrated in Figures 2 and 3 respectively. The figures show the delivery of a block of data composed of four packets from a source node to its k child nodes. In the packet-at-a-time case, the finish times of all the children are nearly equal, while in the block-at-a-time method, the first child has a finish time of

T, the second a finish time of $2 * T$ and so on. From the figures, we see that in the packet-at-a-time case, all children get the block at almost the same time while in the block-at-a-time case, some of the children get the block much earlier.

To the best of our knowledge, there has been no work examining the effect of this link sharing on the data delivery of application-layer multicast trees.[1] In this chapter, we analyze the effect of this link sharing and we demonstrate a simple technique, called Time Division Streaming, to exploit this sharing to reduce the average time to transfer data. We provide analysis of TDS using a simple network model. We then show how the construction of multicast trees in managed overlay networks can take advantage of TDS and propose heuristics for tree construction.Our results show that sending larger blocks of data in multicast trees constructed using our heuristic can provide a substantial improvement in the average finish time of the nodes at the expense of some increase in the maximum finish times of some nodes. We examine the tradeoff in our evaluation of TDS.

The rest of the chapter is organized as follows: In Section 3.2, we describe the link sharing in detail. In Section 3.3, we develop the relation to compute the latency to any host in an end-system multicast tree. We then present algorithms and heuristics to create TDS trees in Section 3.4. We evaluate these TDS trees in Section 3.5, and summarize with some discussion of the results in Section 3.6.

3.2 Time Division Streaming

We present our work in the context of a data delivery application such as an audio or video stream that requires a minimum data rate or bandwidth of b bits per second (bps). We note that this data rate can be achieved either by small packets delivered at regular intervals or by sending a larger block of data, such as a block containing

[1]The authors in [15] point out that the single access link imposes the constraint that the data intended for children of this node must be serialized at the link though they do not consider the problem of scheduling the access to the link.

a few seconds of content, at the beginning of a time period such that the effective data rate over the time period is greater than or equal to b. Thus, in our discussion, we use the term "block" of data to imply the transfer of one or more back-to-back packets of data between nodes in the multicast tree.

We now present a scheme to exploit the serialization of packets at the access link. We call this mode of transfer of data Time Division Streaming. The idea behind this mode of data transfer is essentially to use the available upload bandwidth of a node in a manner similar to Time Division Multiplexing (TDM) of a communication channel, hence the name. In this mode, a node will send a block of data, composed of several packets, to only one of its children, utilizing all of its upload bandwidth for that transfer[2]. Once the transfer of the block is complete, the node sends the block to the next child in order and so on.

Effect of TDS on a single node To illustrate the effect of link scheduling, we revisit our initial example shown in Figures 2 and 3. The figures show a scenario in which we do not consider the propagation delay to the nodes. Let T be the time taken to send a block to a single child using the entire access link bandwidth. Figure 2 illustrates the case where the block of four packets is sent as a packet to each child in turn. The average time to finish receiving the entire block for the children is $3kT/4 + (T/4)(k+1)/2$. Figure 3 shows the case in which the entire block is sent to each child. The average finish time in this case is $T(k+1)/2$. The gain in the average finish time for is $(k-1)3T/8$. More generally, if we let n be the number of packets in a block, s be the size of a packet in bits and B be the uplink bandwidth at the node r, the average finish time, while sending a packet to each child in turn, is $(n-1)ks/B + (s/B)(k+1)/2$ whereas it is $(ns/B)(k+1)/2$ when sending a block at a time. The gain in average finish time over all the nodes is $(s/B)(n-1)(k-1)/2$.

[2]For this work, we assume that the transport protocol used by the application allows for blocks of data to be sent in packets that are back-to-back on the connection.

It is simple to conclude from this analysis that larger blocks are better at reducing the average finish time.

If we incorporate the propagation delay of the links connecting the nodes, the analysis remains unaffected as the propagation delay remains unchanged in both cases with only the transmission delay changing due to the use of larger blocks.

Effect of TDS in a multicast tree To evaluate the effect of using TDS in a multicast tree, consider an interior node r in the tree. If the node has k children, each the root of a subtree, and the children are numbered from 1 to k, consider the finish time of the first child. If this child is the first to receive a block from r, its finish time is ns/B when n packets are sent as a block as opposed to $(n-1)ks/B + s/B$ when a single packet is sent to each child in turn. Irrespective of the scheduling in any other node in this subtree, the finish times of the nodes in this subtree are reduced by this factor of $(n-1)(k-1)s/B$. The finish time of the last child remains unaffected by this and hence, the nodes in its subtree remain unaffected.

In this technique, we are delaying the beginning of transmission to the children that come later in the TDS order to finish transmission of data to the children earlier in the TDS order much sooner than otherwise. This observation shows that if the subtrees of node r are all of equal size, the nodes in child 1's subtree would finish much earlier than nodes in child k's subtree. This can be mitigated if the subtrees of the child nodes were distributed unequally, i.e., if the subtree of the first child were larger than that of the kth child. We build on this intuition in Section 3.4 where we propose tree construction algorithms that take TDS into account.

3.3 Analysis of general case

Until now we have been considering the effect of using TDS to deliver data from a node to its children. We now develop a relation to calculate the finish time of an arbitrary packet in a data stream at any node in a TDS tree. For our model,

Table 2: Summary of notation used

Symbol	Definition
B_a	Upload bandwidth of node a
n	Number of packets in a block
s	Size of a packet in bits
b	Bandwidth required by data stream
m	Packet index in a stream of packets
i	Index of node in a global numbering scheme
$p(i)$	Parent index of node i
$c(i)$	Number of child nodes of node i
$I(i)$	Index of node i in its parent's TDS schedule
$d(i)$	Maximum degree bound of node i

we assume that the data to be transfered is composed of packets of size s. Several packets are aggregated to form blocks. We denote the number of packets in a block by n. Let m be the packet index in a stream of data that is being transfered to the clients.

The end hosts trees have diverse access links connecting them to the Internet [74], varying from dial-up links and cable and DSL modems to ethernet. A characteristic of most of these types of access links is that they offer much larger download bandwidth to the node than upload bandwidth from the node (a factor of 10 with some ISPs using cable modems). Let the upload bandwidth available to each node a participating in the application-layer multicast tree be denoted by B_a (the notation used in this section is summarized in Table 2). The bandwidth requirements of the application creates an upper bound on the number of children that a node can support. This upper bound can be given in terms of the number of children that a node a can support in an application-layer multicast tree and is given by $d(a) = \lfloor B_a/b \rfloor$. We make the assumption that in the tree, nodes with higher upload bandwidth are closer to the source, specifically for a node a with k children $c_0, c_1, \ldots, c_{k-1}$, $B_a \geq max(B_{c_0}, \ldots, B_{c_{k-1}})$. This assumption is reasonable as it has been shown in [45] that to obtain short trees with minimum propagation delay, the nodes with largest degrees should be closest to the source. Given our interest in improving the average latency to transfer data to the clients, any tree considered would have this property.

We begin by considering the simple case with a single source node r with k children. If $0 \leq m < n$ and node i is the j^{th} child of the source, the finish time $t_{i,m}$ of packet m at node i is $t_{i,m} = nj(s/B_r) + (m+1)(s/B_r)$ where the first term represents the time to send the block to the j children before i and the second term represents the time to send the m packets to node i. In general, for $m > 0$, the finish time a packet can be broken up into three parts,

- time to transmit the packets of all previous blocks,
- time to transmit current block to the $(j-1)$ children preceding i,
- time required to transmit the packets of the current block to child i.

This can be represented by $t_{i,m} = (m/n)k(s/B_r) + nj(s/B_r) + (m+1)(s/B_r)$.

For a general application-layer multicast tree, we begin by making the observation that once the first packet of a block arrives at a node, the subsequent packets of that block arrive back-to-back. From our previous assumption about the upload bandwidth of a node being greater than or equal to that of its children, we know that the time to transfer a block to a node is less than or equal to the time that node takes to transfer the block to a child. In other words, once a node begins receiving a block from its parent, it can retransmit the block to its child without waiting for a packet to arrive. Therefore, the time a packet arrives at a node depends on the packet's arrival at the node's parent. We assume that once a node receives the first packet of a block, it immediately begins transmitting the packet to its children. We ignore the propagation delay of links in this analysis to make the exposition clearer but incorporating the delays is straightforward.

Let η represent the block index, i.e., the integer value m/n and let η_1 be the first packet of block η. Let the function $p(i)$ denote the parent of node i and the function $I(i)$, the index of i in its parent's TDS schedule. The time of arrival of a packet m

at a node i can be computed as follows:

$$t_{i,m} = t_{p(i),\eta_1} + I(i)n(s/B_{p(i)}) + (m \bmod n + 1)(s/B_{p(i)}).$$

From the above relations, we note that the latency to any node is dependent on the size of the packet and the number of packets in a block. We evaluate the effect of these factors in Section 3.5.

3.4 Tree Construction

We now consider the problem of constructing trees that take into account TDS. Current algorithms for constructing application-layer multicast trees optimize for delay or bandwidth without considering the transmission delay at interior nodes. We wish to create trees that not only take into account the transmission delay but also optimize for the block size being used in TDS. We begin by stating the objective for our tree construction algorithm.

The optimization problem can be stated as follows: Let $G = (r, N, E)$ be a complete graph with a source node r and end hosts N. Let the degree constraints of the nodes be given by the vector D. Let E be the set of edges between the nodes. Our objective is to find the tree T with minimum average finish time to transfer the block B and satisfies the degree constraints. We first consider the case where the end-to-end delay between any pair of nodes to be the same (in this case zero), i.e., we ignore the propagation delay but not the transmission delay caused by the link scheduling. We present a centralized algorithm in Figure 4 that constructs an optimal tree for this case. The algorithm is run by a designated node such as the source in the following manner: First, the nodes are sorted in non-increasing order of their degree constraints. In the main loop, the next node from the sorted list is selected and attached to the tree at the position with the minimum finish time until all nodes are attached. If no attachment points exist due to the degree constraints of the nodes, the tree returned is empty.

```
Tree T = createTree(Nodes N, Source s, DegreeConstraints D,
                    Blocksize B)
  Sort the nodes of N in non-increasing order of degree
    constraints into n_1,n_2,...,n_|N|.
  T = {s}
  Compute t(s) as the time to transmit B to first available
    child of s.
  Insert s into MinHeap with value t(s).
  i = 1
  while i <= |N|
    Get next node a from MinHeap.
    Attach n_i as child of a.
    T = T ⋃ {n_i}
    if c(a) < d(a)
      Recompute t(a) and insert a into MinHeap with
        value t(a).
    if d(n_i) > 0
      Compute t(n_i) and insert n_i into MinHeap with
        value t(n_i).
    i++
  done
  return T
```

Figure 4: Tree Construction Algorithm for TDS ignoring propagation delays

Theorem 1. *The algorithm createTree generates a tree such that the nodes have the minimum finish times given the degree constraints D.*

The proof relies on the following lemmas.[3]

Lemma 2. *For any node in the optimal tree, the size of its childrens' subtrees are in the order in which data is sent to the children, i.e., if data is sent to child i before child j, the size of i's subtree is greater than or equal to j's subtree.*

Lemma 3. *For any node in the optimal tree, the child with the larger degree bound is sent data before a child with a lesser degree bound.*

A corollary to lemma 3 is that for any node, the child with the largest degree bound is the first to which data is sent.

[3]The complete proofs of the lemmas can be found in Appendix A.

Lemma 4. *Given a set of N nodes with degree constraints, the optimal tree has the node with the maximum degree constraint as the root.*

The sketch of the proof of lemma 4 is as follows: We assume, for contradiction, that the optimal tree does not have the node with the maximum degree constraint at the root. It follows that for some subtree in the optimal tree, the node a with the maximum degree constraint is the child of a node with a smaller degree constraint. By lemma 3, a is the first child to be sent data by its parent. We show that exchanging a with its parent and rearranging the subtrees of a and the subtrees of the parent such that the larger subtrees are attached to a leads to a tree with a lower finish time, violating our assumption that this tree was optimal.

The proof of lemma 3 is very similar.

Proof of Theorem 1. We prove this by induction on i, the number of nodes attached to the tree. If $i = 1$, then there is only one node in the tree, the source s and it is clearly optimal. Assume that the algorithm creates an optimal tree for i nodes. By the algorithm, the degree constraint of the $i + 1$st node is less than or equal to the degree constraint of any node in the tree uptil now. By lemma 4, node $i+1$ can only be attached as a leaf, and the position that minimizes the finish time of $i + 1$ is the position with the minimum transmission delay from the source to $i+1$ which is node a from the algorithm. Thus, the tree with $i + 1$ nodes is also optimal. □

The general problem of constructing optimal trees with non-uniform propagation delays between nodes has been shown to be NP-Hard in [9]. To handle tree construction for TDS taking propagation delays into account, we propose the following *TDS heuristic* that attempts to balance the degree bound of a node and its propagation delay to the tree. The heuristic iteratively adds nodes to the tree in the following manner: Initially, three sets of nodes are created, N_A consisting of nodes that are attached to the tree, N_{Av} consisting of nodes that are attached and can accept more

children and N_U consisting of nodes that are unattached. Let $l(u)$ be the latency of node u to the source along the tree. In the beginning, N_A and N_{Av} contain only the source node and N_U contains all other nodes. At each iteration, the algorithm computes a cost for each $v \in N_U$ as

$$Cost(v) = \min_{u \in N_{Av}} \alpha l(u)/l_{max} + \beta d(v)/d_{max} + (1-\beta)\delta(u,v)/\delta_{max}$$

where $l_{max} = max_{u \in N_{Av}}\, l(u)$, $d_{max} = max_{u \in N_U}\, d(u)$ and $\delta_{max} = max_{u \in N_U, w \in N_{Av}}\, \delta(u,w)$ are normalization constants.

The variables α and β control the weight of the different factors in the computing the cost of each unattached node. The value of α controls the extent to which the latency in the tree to the attachment point affects the cost, while β determines the relative importance of the degree constraints and the propagation delay of the unattached nodes. From our evaluation, we observed that the heuristic is relatively unaffected by the value of α and so we fixed the value of α to be 1. We explore the effect of the β parameter on the average latency in the next section.

3.5 Evaluation

3.5.1 Methodology

We used libraries provided by the *p-sim* simulator [50] to write our simulation. For our simulations, we begin by creating a representative Internet topology using GT-ITM [16] comprising of 4050 nodes. We then randomly choose some of the nodes to be the hosts participating in the application-layer multicast tree. We randomly select one of the nodes to be the source of the end-system multicast. The degree constraints for the nodes are assigned from a uniform distribution with the source node being assigned the maximum degree constraint. The link latencies are drawn from uniform distributions with [50ms, 200ms] for the transit links, [25ms, 100ms] for the transit-stub links and [5ms, 50ms] for the stub-stub links. The stream bandwidth requirements are set at 8kBps per child in the multicast tree. We construct the

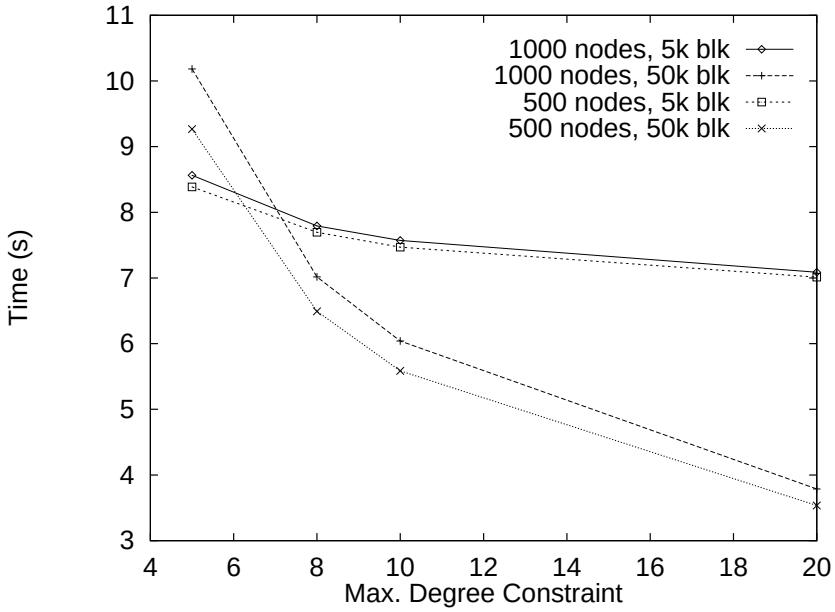


Figure 5: Varying the maximum degree constraint of the nodes participating in the multicast tree

application-layer TDS multicast tree using the algorithm detailed above. For all the experiments we report the average finish time as the time to transfer 50 kB of data from the source to all the nodes in the tree. We chose block sizes of 50kB and 5kB as reasonable bounds on the size of an application's data unit. The packet size used is usually 500 bytes. We also experimented with 1500 byte packets but the results were similar with the 1500 byte packets having a slightly larger average finish time. We begin by investigating the different parameters that affect the TDS scheme.

3.5.2 Effect of TDS parameters

In Figure 5, we plot the average[4] finish time of the nodes in the TDS tree on the y-axis against the maximum degree constraints allowed for the nodes on the x-axis. Each line represents different size trees with varying block sizes. From the graph, we see that for smaller degree constraints, the smaller block sizes are better for TDS. The small degree constraint results in trees that are tall and narrow, resulting in poor

[4]In all cases, the median was less than the average. We omit plotting the medians for clarity.

Table 3: Node distribution of TDS trees for different block sizes.

Max. Degree Constraint	Block size (kB)	Percentage of nodes in first two subtrees	Depth
10	50	66	8
	5	39	5
8	50	66	8
	5	39	6
5	50	75	9
	5	67	7

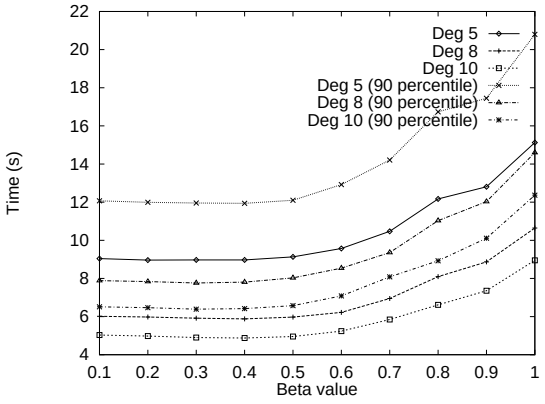


Figure 6: Varying β with block size of 50kB

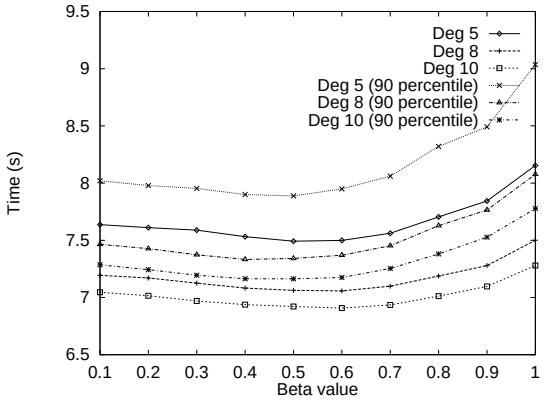


Figure 7: Varying β with block size of 5kB

performance of TDS as the difference between the finish times of the first and last child at a node are not significant enough to offset the longer transmission delays. As the maximum degree constraint is increased, the trees created are wider and the larger block size has significantly better performance. The trees for the larger block size are more unbalanced with the subtrees of the children that are earlier in the TDS order being much larger that the subtrees of those later in the TDS order. This can be seen in the table in Table 3 in which we show the size of the subtrees in terms of the percentage of the total nodes that the subtree contains. Although the degree bounds for the nodes are assigned from a uniform distribution, the distribution of the degrees of nodes in the final tree is similar to the distribution of degrees observed by Sripanidkulchai et al [74].

3.5.3 Effect of β parameter on the TDS heuristic

In Figure 6, we plot the average finish time of the nodes in the TDS tree on the y-axis against various values of β on the x-axis. Each line represents trees constructed

with a particular maximum degree constraint and each point is the average of five runs of the simulator with different seeds. We observe that the average finish time is only marginally affected for values of β up to 0.5. Actually the average finish time is reducing in this interval, but as the value of β increases above 0.5, the average finish time increases quickly. This is also seen in Figure 7, in which the curves are plotted for a block size of 5kB. These plots show that selecting the nodes primarily on the basis of the propagation delay to construct TDS trees results in poor performance. The best balance seems to be to equally weight the degree constraints of the nodes and their propagation delay when considering the next node to add to the tree.

We also plot the 90^{th} percentile value of node finish times for each of the degree constraints. We observe that the 90^{th} percentile value of the 50kB block with a degree constraint of 10 has a smaller finish time than the average finish time using 5kB blocks. This indicates that most of the nodes benefit when we use larger blocks. Another observation we can make from the graph is that the 90^{th} percentile value is closer to the average finish time for the 5kB block size than for the 50kB block size, indicating the increased variance due to the larger block size.

Planet-Lab experiments To evaluate the effects of TDS in a real-world scenario, we developed a small application to run on the PlanetLab network [58]. In a limited experiment using 16 nodes, block sizes of 50kB and 5kB, a packet size of 1000 bytes, we observed that the average finish time of the 50kB block size was 6.32 seconds while for the 5kB block was 8.41 seconds which agrees well with our analysis.

3.5.4 Performance relative to existing heuristics

There have been other heuristics proposed to construct degree-bounded trees for application-layer multicast. The heuristics proposed are for minimizing the maximum latency to clients [69] (which we call Compact Tree) and for minimizing the cost of using proxies [45] (which we call Min Cost). The Compact Tree heuristic

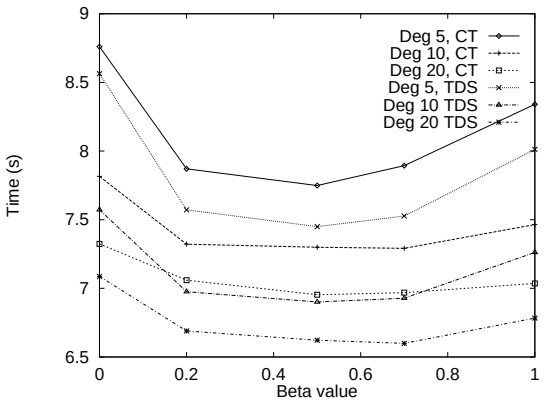


Figure 8: Varying β with block size of 5kB using CT and TDS heuristics for 1000 nodes

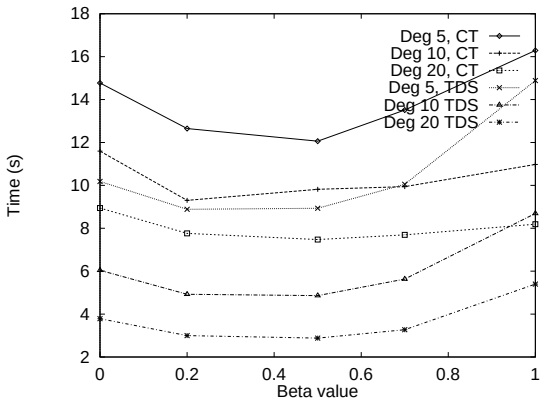


Figure 9: Varying β with block size of 50kB using CT and TDS heuristics for 1000 nodes

incrementally constructs a minimum spanning tree from the source s. For each node v not in the tree, it finds the minimum cost edge (u, v) from a node u in the MST. The cost that is minimized is the overlay delay $\delta(s, v)$ from the source to the node v. The Min Cost heuristic is quite similar except for the cost function used to select the next node. The Min Cost heuristic considers the minimum latency edge (u, v) as well as the degree constraint d of the nodes while selecting the best node to attach to the tree. The cost function is $\gamma_\alpha(v) = \alpha d(v)/d_{max} + (1-\alpha)\delta_{min}/\delta(s, v)$. The α parameter plays the same role as the β parameter in the TDS heuristic and d_{max} and δ_{min} are normalization constants. Both heuristics do not consider the cost of transmission of data while constructing the trees.

For our simulations, we implement both heuristics using the same routine. The value of the parameter $\beta = 0$ creates trees based on the Compact Tree heuristic while other values of β create trees based on the Min Cost heuristic. We plot the average finish times for delivering 50kB of data using trees with 1000 nodes constructed by the different heuristics in figures 8 and 9 for block sizes of 5kB and 50kB respectively. In general, the graphs show that the TDS heuristic performs much better for every degree bound that is used as it considers the transmission delays incurred at each node. The magnitude of the improvement of the TDS heuristic is larger with block

sizes of 50kB than with 5kB. The trees created by the TDS heuristic for the 5kB blocks are not very different from the trees created by the other heuristics and so the improvement seen is on the order of a second in the average finish times. On the other hand, the trees for the 50kB block size created by the TDS heuristic exploit the larger block size to create trees that are very different from those created by the other heuristics resulting in significant improvement over the other heuristics. When β is in the range of 0.5 to 0.7, the Min Cost and Compact Tree heuristics perform best as they consider a combination of the degree constraints along with the propagation delay to the nodes in the tree.

3.6 Discussion

Our results show that nodes sending large blocks of data to each child in turn can reduce the average finish time of nodes in the multicast tree. The tradeoff involved in this gain is the increased variance of the actual finish times of nodes. Based on this tradeoff, the block size and packet size for TDS can be specified to match application requirements. For example, the increased variance with larger block sizes can be used as an incentive mechanism to encourage nodes to dedicate more uplink bandwidth to the application. This in turn would place those nodes in positions where their finish times are earlier.

In this chapter, we examined the effect of the single access link that many end hosts that participate in application-layer multicast have. We showed that the average finish times of nodes in the tree are affected by the way in which this link is used to transfer data to a node's children. We proposed a technique called Time Division Streaming to share this access link such that the average finish times are reduced as compared to previous work. We also provided analytical results based on a limited model of this technique and propose heuristics that take this serialization into account when constructing the tree.

Using the TDS heuristic to construct multicast trees in managed overlays, we showed significant reduction in the average finish times of nodes. The heuristic exploits the effect of TDS by creating trees such that the interior nodes have unequal subtrees with the subtrees of children earlier in the TDS schedule being larger.

CHAPTER IV

ROUTESEER: AN OVERLAY NODE PLACEMENT ALGORITHM

4.1 Introduction

Service overlay networks [23] are composed of dedicated nodes that connect to the overlay for long periods of time. These service overlays, hereinafter referred to simply as overlays, are generally designed to support a particular service objective. Overlay network design includes the problems of node placement and selection of overlay links between nodes.[1] Where necessary or appropriate, the design may include on-line mechanisms to dynamically modify the links in response to changing network conditions or client behavior [24]. Nodes may also be added or removed, however that is likely to occur on fairly long time scales in a service overlay. The addition of a node may, for example, require contractual negotiations with an underlying network service provider and/or a co-location facility.

Much prior work has been devoted to the problem of link selection and network maintenance, with an assumption that the overlay nodes are given (e.g., [23, 14, 60, 9]). In contrast, we are interested in the node placement problem. Node placement clearly constrains the ability of the overlay to meet a particular performance objective, regardless of how well links are chosen. Further, we focus on the design objective of network resiliency, the ability of the overlay network to mask failures in the network links. These failures may be hard, i.e., links that go down, or soft, i.e., links that suffer periods of very poor performance.

Our decision to focus on network resiliency derives from the following observations.

[1] In contrast, peer-to-peer overlays have little or no control over the placement of the nodes.

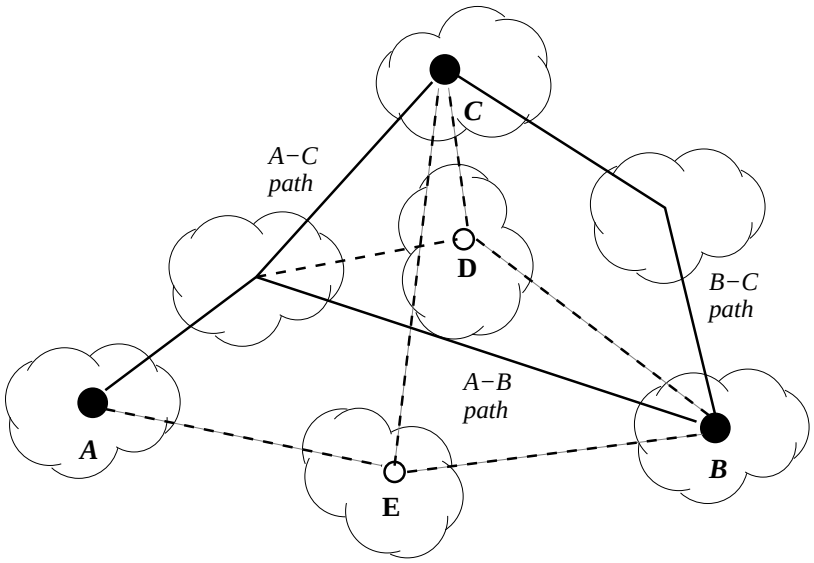


Figure 10: Example of intermediate node placement

Service overlays are compelling because they improve upon the service offered by the underlying native network. The ability to mask underlying network failures is a useful service in its own right, as well as the basis for allowing overlays to provide more sophisticated services such as SureRoute [3]. The increasing use of multi-homing [4] provides evidence that clients are willing to pay for increased resiliency as part of improved performance.

We split the problem of overlay node placement into two parts: the first is to place overlay nodes called *client proxies* "close" to the clients. For this we use existing proposals for placing nodes or services in proximity to clients [40, 12]. The second part of the problem is the placement of other overlay nodes such that links between client proxies are protected, i.e., alternate disjoint overlay paths exist for each direct path between client proxies. This is illustrated in Figure 10 which shows a small overlay network. Nodes A, B and C are client proxies and the direct paths between nodes A, B and C are shown using solid lines while the dotted lines indicate overlay paths. Using node D as an intermediate node provides protection to the direct path between B and C but not to paths A–B and A–C as the path A–D shares common

links with both paths. On the other hand, using node E as an intermediate node provides protection to all paths between A, B and C.[2]

One approach to identifying intermediate nodes involves active measurements between potential node locations, with the objective of minimizing the number of shared network links [35, 18]. Based on measurements, a set of intermediate node locations can be chosen. Naturally, these proposals require active measurements to identify node locations. More significantly, they require that the overlay network provider already have access to the full set of potential node locations, since this is needed to conduct the measurements.

In this chapter, we propose a technique called RouteSeer to select locations for overlay nodes. RouteSeer is based on the simple idea of examining routing tables at a few locations in the network to place the nodes. Our technique can examine the entire network topology, rather than a limited set of locations, for potential overlay node locations while requiring much less active probing of the network than previous proposals. We first examine the information provided by routing tables at the set of client proxies of the service overlay. Under certain assumptions about network routing, we show how this information can be used to select overlay node locations that minimize the sharing of network paths. We then extend this method to more realistic networks by relaxing our assumptions about ideal network routing behavior and incorporating some limited active probing of the network. Our simulations indicate that we can achieve significant improvements in creating overlays with non-overlapping network paths using our technique.

The rest of the chapter is organized as follows: In the next section we discuss the background of the overlay node placement problem and define the problem more formally. In Section 4.3, we explain the working of RouteSeer. We evaluate the

[2]We ignore for now the issue that E could become overloaded if there are multiple direct path failures to mask.

RouteSeer algorithm in Section 4.4 using simulations on a wide variety of topologies. We present some related work in 4.5 and conclude in and summarize in Section 4.6.

4.2 Problem Formulation

A general Overlay Network Design problem can be stated as follows: Given an undirected graph $G = (V, E)$ representing the underlying network, a set of clients $C \subset V$, find a set $O \subset V$ of overlay nodes such that a metric over all possible $i, j \in C$ and $k \in O$ is optimized. We are interested in the ability of the overlay to mask failures in underlying network links. Using terminology from [35], we define the *direct path* between $i, j \in C$ as the shortest path between i and j and an *indirect path* between i and j using k as the direct path from i to k and the direct path from k to j for some $k \in O$. We say an overlay path between nodes i and j is *resilient* if there exists an indirect path through some node $k \in O$ which shares no common network nodes with the direct path between i and j. The overall overlay resiliency can be defined as the total number of overlay paths that are resilient. Another useful metric of overlay resiliency we use is the total number of vertices that are common to the direct and indirect paths for each pair of overlay nodes. We will formally define the problems based on these overlay resiliency metrics next.

As we discuss in Section 4.3, it is helpful to fix the locations of some of the overlay nodes based on the locations of the clients of the overlay service. Let C be the set of overlay nodes whose locations are fixed. We call these nodes *client proxies* as they effectively act as proxies for the clients that connect to them.

We can now formally define the problem in the following manner: Given an undirected graph $G = (V, E)$ and a set $C \subset V$ of client proxies, define $S(i, j)$, $\forall i, j \in V$, to be the set of vertices that lie on the shortest path from i to j. Define $overlap(i, j, k) = |(S(i, j) \cap S(i, k)) \setminus \{i\}| + |(S(i, j) \cap S(k, j)) \setminus \{j\}|$ $\forall i, j \in C$ and $k \in V$. Intuitively, overlap counts the number of common network vertices in the

direct path between i and j and the indirect path using an intermediate overlay node k. The Minimum Overlap problem can then be stated as finding the set of intermediate overlay nodes of size n that minimizes the total overlap across all pairs of client proxies. More formally,

Minimum Overlap Problem: Select $O \subset V$, $|O| = n$, such that

$$\sum_{i,j \in C, i \neq j} \min_{k \in O} overlap(i, j, k)$$

is minimum over all possible sets O of size n.

The above formulation requires that we know the overlap for each possible element of O to compute the minimum possible overlap. In practice, this could require knowledge of the paths between every node in the graph, which in turn requires complete knowledge of the graph topology. Therefore, we define the following problem which is similar to the Minimum Overlap problem but which can be solved in practice using only routing table information available locally at each client proxy, without knowledge of the entire graph topology.

Maximal Disjoint Path Problem: Let $I\{\}$ be an indicator function with $I\{a\} = 1$ if a is true and 0 if a is false. Select $O \subset V$, $|O| = n$, such that

$$\sum_{i,j \in C, i \neq j} \min_{k \in O} I\{overlap(i, j, k) = 0\}$$

is minimum over all possible sets O.

The Maximal Disjoint Path problem tries to find a set of intermediate overlay nodes of size n that maximizes the number of client proxy pairs that have a resilient overlay path. In the next section, we outline our heuristic which solves the Maximal Disjoint Path problem for the AS graph.

4.3 RouteSeer

As discussed in the previous section, solving the Minimum Overlap problem requires knowledge of the complete network topology which is usually not available. Active

probing of all possible locations for the intermediate nodes has a prohibitive cost and is not practical in most scenarios. In this section, we describe RouteSeer, a technique to place intermediate overlay nodes for service overlay networks using routing table information. The RouteSeer algorithm is designed to solve the Maximal Disjoint Path problem outlined in the previous section, rather than the Minimum Overlap problem, because we limit our algorithm to use only information locally available to the client proxies. We show in our evaluation that, in practice, the RouteSeer algorithm gives good solutions to the Minimum Overlap problem as well.

Until now, we have been using the term "node" in a generic sense as vertices in a network graph. The overlay nodes are, of course, machines connected to the Internet. The placement problem actually has two parts, deciding which Autonomous Systems (ASes) to locate the nodes in, and then deciding where in the AS to place the node. We will primarily focus on the first part, i.e., selecting the ASes in which to place the intermediate overlay nodes. Our heuristic could be extended to work in the intra-AS case as well.

For the remainder of this chapter, we will assume that the network graph is the Autonomous System (AS) graph of the Internet, that the nodes represent ASes, and the paths that we refer to represent AS paths.

The intuition behind the RouteSeer algorithm is quite simple: if we consider the shortest path tree constructed with a particular node as the source, once a particular outgoing link is chosen for a path p_1 to a destination d_1, any shortest path p_2 to a destination d_2 using a different outgoing link will not intersect the path p_1, provided consistent tie breaking is used for equal cost paths. Clearly, this requires the client proxies have multiple (> 1) egress links to ensure that the outgoing paths are disjoint. In the case where a node has a single egress link, the paths can be made disjoint up to that link, but that link would remain as a single point of failure.

4.3.1 The RouteSeer Algorithm

For any service overlay network to be useful to clients, the placement of the overlay nodes should reflect the locations of the clients, i.e., nodes providing service to the clients should be close to them. Client nodes can be transient in that they can connect to the overlay network for short periods of time. This behavior must be taken into consideration when placing the overlay nodes, which are assumed to be long-lived. For this reason, RouteSeer takes a three-step approach to placing overlay nodes in the network.

4.3.1.1 Client Proxies

In the first step, we place special overlay nodes, called Client Proxies or CPs, close to the clients of the overlay network. Close could be in terms of the number of hops, latency or any metric that the overlay application requires. The main function of these client proxies is to act on behalf of the clients that connect to them. Ensuring that the CPs are close to the clients improves the performance perceived by the clients. The use of client proxies frees the clients from running any routing protocol to discover the routes to destination nodes, as the client proxies can perform this function for them. The client proxies are assumed to be long-lived and so the placement of the other overlay nodes can be better optimized for the set of client proxies and thus, for the clients as well. Using client proxies, the problem space can also be reduced from finding resilient paths between each pair of clients to finding resilient paths between each pair of client proxies. For the purposes of our heuristic, we assume that the client proxies have already been placed based on various methods such as those proposed by Krishnamurthy and Wang [40] or Barford et al. [12].

4.3.1.2 Get potential locations

RouteSeer next attempts to place intermediate overlay nodes to provide path diversity for the paths between client proxies. We begin by describing a simplified version of

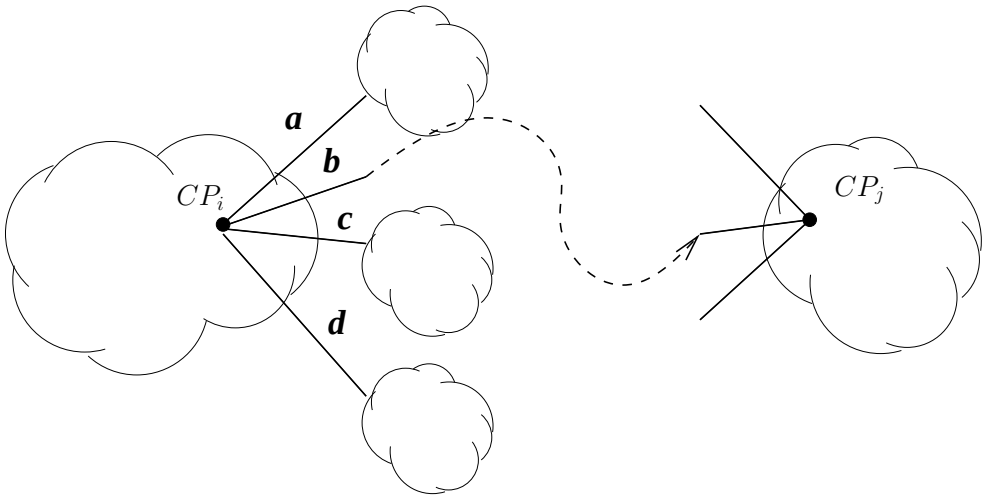


Figure 11: Path between client proxies CP_i and CP_j

RouteSeer, for which we make the following two assumptions:

1. The network layer uses shortest path routing when forwarding packets.

2. Network paths are symmetric, i.e., the path from a node i to a node j is the same as the path from j to i.

We will later relax these assumptions and explain the modifications required. Within the main algorithm, we use a *reachability table* generated from the forwarding table where the entries are of the form $\langle nexthop, addressprefix \rangle$. Each entry in the table contains the set of address prefixes that are reachable from that node using the particular link.

The main algorithm is as follows: Consider a pair of client proxies CP_i and CP_j as shown in Figure 11. We construct reachability tables for both nodes from the respective forwarding tables. Assume that traffic from CP_i destined for CP_j is forwarded along link b. This information can be found from the forwarding table for CP_i. If we want to achieve overlay path resiliency for the path from CP_i to CP_j, clearly the path to the overlay node that acts as a relay cannot be along link b, i.e., the intermediate overlay node cannot be in any of the address prefixes for which CP_i would use link b to forward the packets. Using the reachability table, we can eliminate all such address

prefixes by removing the entry for link b. We call the set of address prefixes that remain V_{ij}. By our first assumption of shortest path routing, any path that begins with link b will not intersect any path beginning with a link other than b. Thus, we can be assured that if the overlay node is located in an address prefix that belongs to V_{ij}, the path from CP_i to the overlay node will not intersect the path from CP_i to CP_j. For duplex communication, we need to ensure that the reverse path from CP_j to CP_i should also not intersect the path from the overlay node to CP_i or the path from CP_i to the overlay node. By our second assumption of path symmetry, CP_i receives traffic from CP_j on the same link b on which it sends traffic to CP_j, and so the paths would be link-disjoint. Node CP_j can compute a corresponding valid set V_{ji} based on information in its forwarding tables in the same manner.

The intersection of the sets V_{ij} and V_{ji}, $A_{ij} = V_{ij} \cap V_{ji}$, gives the set of address prefixes in which the overlay nodes that provide path resiliency for the path from CP_i to CP_j could be located. It is possible to have A_{ij} or one of V_{ij} or V_{ji} be empty in which case the path between this pair of client proxies cannot have a resilient overlay path between them.

When considering the sets of address prefixes to create the intersection, we restrict the prefix length to be at most 24 bits. The primary reason for this is to limit the number of address prefixes to consider to a manageable size. In a recent BGP routing table dump, the number of prefixes with length greater than 24 bits was approximately 1% which also indicates that this prefix length is a good number to choose. We ignore any prefixes longer than 24 bits in the intersection even if such prefixes exist. In case of address prefixes that overlap, we take the longest matching prefix that is in both sets to be the result of the intersection, e.g., if V_{ij} contains the prefix 192.168.0.0/16 and V_{ji} contains 192.168.0.0/20, then their intersection A_{ij} will contain 192.168.0.0/20 which is the longest matching prefix that matches both address prefixes.

```
computeLocations(GAT, K)
S_A = ∅
while | S_A | ≤ K do
  Sort the rows of GAT according to the counts
  for each row.
  Assign max ranked address prefix from S to p
  S_A = S_A ∪ { p }
  for each CP pair cp for which p is a
  potential location
    decrement all other address prefixes which
    have this path by removing cp from their
    set in the GAT
  end
  return S_A
end
```

Figure 12: Heuristic for computing K potential overlay node locations

Once the set A_{ij} is created for a pair of client proxies CP_i and CP_j, we store this information in a table called the Global Address Table (GAT). This table has a row for each possible address prefix and each row stores the set of CP pairs for which this prefix appears in the intersection set. For example, if the prefix 192.168.0.1/24 appears in the set A_{ij}, then we add the pair (i, j) to the set of pairs stored in the row corresponding to 192.168.0.1/24 in the GAT. Thus, this table tracks the set of client proxy pairs for which a particular address prefix is a potential overlay location. This table also stores address prefixes at the granularity of 24-bit prefixes. If a particular address prefix spans multiple /24 address prefixes, all rows are modified.

4.3.1.3 Compute feasible locations

This procedure of creating the set A_{ij} and adding to the GAT is performed for all possible pairs of client proxies i and j. At the end of this process, each entry in the GAT contains the number of paths for which the address prefix is a potential location of an overlay node. From the 2^{24} available address prefixes, we need to extract a set of K address prefixes that can provide path resiliency to all pairs of client proxies. This problem is exactly the Minimum Set Cover problem, with the set S of all pairs

of client proxies which is to be covered. Each address prefix covers a subset of S (the set of paths stored in the corresponding row in the GAT) and our objective is to find a cover of S which is at most of size K. This problem is known to be NP-complete [28]. We use the greedy heuristic outlined in Figure 12 to extract a set of K address prefixes that cover the maximum number of client proxy pairs.

The algorithm maintains a set of accepted address prefixes S_A, which is initialized to null. The address prefixes from the GAT form the available address prefixes from which the next is to be selected. The algorithm works in the following manner: iteratively, the address prefixes in the GAT are ranked in order of the number of client proxy paths for which the address prefixes are potential intermediate node locations, i.e., the size of the set of CP pairs in each row. The highest ranking address prefix p is removed from the GAT and added to S_A. Since p is a potential location for a set of client proxy pairs which no longer have to be considered by the remaining address prefixes, all remaining address prefixes that also are potential locations for these pairs have their counts decremented by removing the client proxy pairs from the corresponding rows in the GAT. This process is repeated K times to extract K address prefixes.

This set of address prefixes returned by the algorithm may not provide path resiliency to all client proxies, but we impose this limit of K based on the number of potential locations required by a particular overlay network. In our evaluation, we show that, in practice, this limit K can be quite small.

Note that the procedure considers all paths between the client proxies to be equal. The same algorithms can be run in scenarios where the paths have different weights corresponding to their priority in the overlay. The only difference is that when the address prefixes in the GAT are being updated, instead of incrementing by one, the address prefixes can be updated using the weight of the path.

In situations in which load on intermediate nodes is of concern, we can modify

the algorithm in the following manner: When the highest-ranked address prefix p is removed from the GAT, instead of removing *all* pairs for which p is potential location, we remove only m CP pairs. The value m may depend on the type of nodes deployed or be an application-dependent value. The remaining CP pairs remain in the GAT to be protected by some other address prefix.

The final set of address prefixes returned by RouteSeer can be used to place the overlay nodes or they can be used as the starting point for further probing to incorporate other performance metrics in the overlay paths such as latency and bandwidth.

4.3.2 Effect of Assumptions

The above discussion of RouteSeer has been in the context of the assumptions made in Section 4.3.1.2. We now discuss the effect of removing those assumptions one-by-one. First, we relax the second assumption of symmetric paths, since paths on the real Internet are known to contain asymmetries.

If the path taken by traffic from CP_i to CP_j is different from the path taken by traffic from CP_j to CP_i, a problem arises in the second step of RouteSeer. When the link b and its associated address prefixes are eliminated from consideration for V_{ij}, it was under the assumption that the reverse traffic from CP_j also traverses the same link. If this is not assured, we need to add another mechanism to discover the link that the reverse traffic would use. To do this, both nodes CP_i and CP_j perform traceroutes to each other and exchange the results. By examining the traces, CP_i can discover the link through which the traffic from CP_j reaches it. Assuming that this is link c, we now eliminate both links b and c and their associated address prefixes from inclusion into V_{ij}. Similarly, V_{ji} would eliminate the links on which it sends traffic to CP_i as well as the link from which it receives traffic from CP_i. The remainder of the RouteSeer algorithm proceeds as before.

The first assumption made was that shortest path routing was used in the network.

This assumption is violated in the Internet due to the use of policy routing [80]. We note that with this assumption, the locations returned by RouteSeer ensure that the path resiliency provided is complete, i.e., no intermediate links are shared between the direct and indirect paths. If this assumption is violated, it is likely that some of the overlay paths would share links with the direct path between client proxies. We evaluate the extent to which this happens in the next section.

4.4 Evaluation

4.4.1 Applicability of RouteSeer

Currently, there are few commercial service overlays deployed in the Internet and only a few research testbeds. It is difficult to predict the eventual size of these networks when they are deployed. To get an estimate of how many client proxies we should consider, we looked at some examples of such networks. Research networks like the RON [6] testbed have approximately 17 nodes, and the NLANR Web proxies [55] also have a similar number deployed. On the other hand, Akamai, whose network of content servers could be considered to be a service overlay, has approximately 15K servers deployed. Clearly, with such a density of nodes, placement of nodes for network resiliency is of less importance; there is a high likelihood that an intermediate node exists for almost every path. A likely scenario for deployment would be a network like Planet-Lab, which is composed of approximately 400 nodes at various locations across the globe. Thus, in our evaluation, we look at small (10 nodes) to medium (500 nodes) sized networks.

A concurrent question is the number of intermediate overlay nodes that need to be deployed for the client proxies. Note that the clients of the overlay would connect to the client proxies and these intermediate nodes that are to be placed using RouteSeer are, in some sense, "overhead". Thus, it is reasonable to conclude that the number of intermediate nodes, depending on the performance that is desired, would

be quite small. With that in mind, we perform our evaluations with the number of intermediate overlay nodes ranging from 2 intermediate nodes to 50 intermediate nodes for the larger networks (i.e., 10% of the number of client proxies).

4.4.2 Methodology

All our simulations were performed using a simulator built with the help of libraries from the *p-sim* simulator [50]. The simulator takes as input a network topology and parameters for the number of intermediate overlay nodes required and the number of client proxies in the network. It then places the client proxies randomly in the network topology[3] and computes the routing tables for each proxy using Dijkstra's shortest path algorithm and a simplified model of policy routing described in Section 4.4.3. The simulator then proceeds to place the required intermediate overlay nodes in the network based on the specified placement scheme and evaluates the overlap that results.

For many of the experiments discussed below, we use AS network topologies generated from the RouteViews project [61] which has BGP peering sessions with many ASes in the Internet. We discuss the dataset used in the Internet experiments in Section 4.4.7.

We evaluate the performance of the RouteSeer algorithm with respect to a random placement of the intermediate overlay nodes and to a placement scheme suggested by Han et al. [35] for router-level overlay node placement. To implement the Han scheme, we first randomly select a large pool of nodes that is at least twice the number of required intermediate overlay nodes. We then run the clustering-based heuristic proposed in [35] to create the required number of clusters. From each cluster, we select one node at random as the representative of that cluster. This gives us the required number of intermediate overlay nodes.

[3]We could have restricted client proxy locations in some fashion (e.g. stub ASes) but we chose to be non-restrictive for this study.

4.4.3 Policy Routing

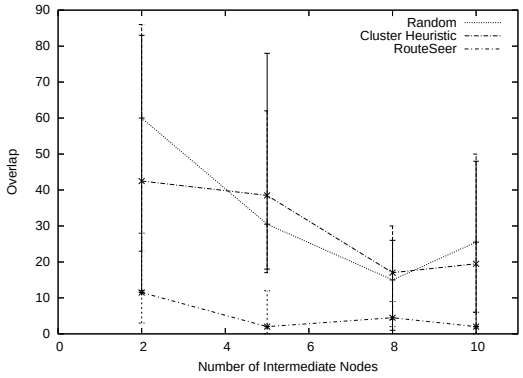
An assumption made initially while describing the RouteSeer algorithm was that routes leaving a node through a particular link would not intersect routes leaving through other links, i.e., shortest path routing is being used. Routing on the Internet is affected by policies that individual ASes apply to routing packets through their domain [80], and this results in paths that do not necessarily follow the shortest path through the AS graph. To have a more robust evaluation of RouteSeer, we need to quantify the effect of routing policies on the algorithm's operation.

There is a large body of work on understanding and measuring the effects of routing policies in the Internet. We attempt to recreate the effect that routing policies would have on routing in our simulation. For this, we use work done by Gao [27] on identifying AS relationships to mark the edges between ASes as either customer-provider, provider-customer or peer-peer edges. We also use a simplified version of work by Subramanian et al. [78] on modeling AS paths to define routes on this annotated AS graph. We only allow paths of the form [(customer-provider)*, (peer-peer), (provider-customer)*] where the '*' allows for multiple instances of a particular type of an edge. This simple model allows us to impose some basic policies on the paths in the AS graph. Note that this model does not cover all possible routing policies and must be viewed as a simple approximation.

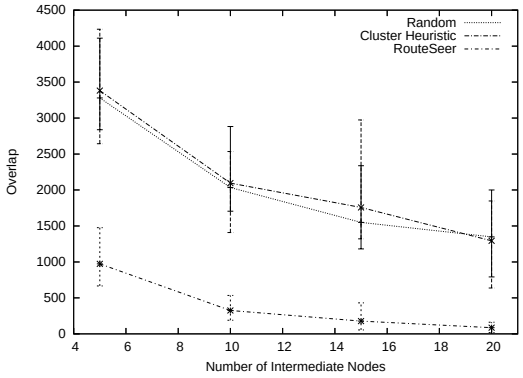
4.4.4 Evaluation on the AS topology

In our first set of simulations, we compare performance of RouteSeer, the clustering heuristic, and Random placement over the AS topology. In Figures 13(a) to 13(c), we plot the performance of the three placement algorithms as we vary the number of intermediate nodes used from 2 to 50. For the 10 client proxies case, we restrict the number of intermediate nodes to a maximum of 10 nodes.

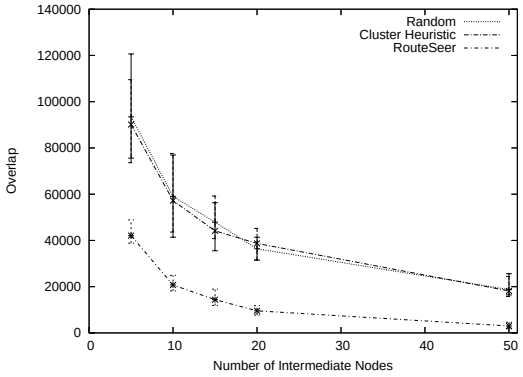
We plot the number of intermediate nodes used on the x-axis and the number of



(a) 10 Client Proxies with Policy Routing



(b) 100 Client Proxies with Policy Routing



(c) 500 Client Proxies with Policy Routing

Figure 13: Overlap using RouteViews dataset

overlapping nodes on the y-axis. The smaller the overlap, the better the performance of the particular set of intermediate nodes. For each value of the intermediate nodes, we repeat the simulation 10 times with different seeds, effectively trying 10 different sets of client proxies. We show the minimum, median and the maximum values of the 10 trials in the graphs plotted. From the figures, we see that the placement due to the RouteSeer algorithm is consistently better than the cluster heuristic. As more overlay nodes are used, the overlap for all placement schemes reduces, primarily due to the increased number of choices for each path that more intermediate nodes offer. We observe that the overlap due to Random placement is very close to that of the Cluster Heuristic. We conjecture that this is because the quality of the nodes selected by the Cluster Heuristic is completely determined by the initial set of nodes on which it is run.[4] Since the initial set of nodes is selected at random, the performance of this heuristic is close to that of random placement. Also, as we increase the number of overlay nodes, the ratio of nodes selected by the heuristic from the initial random set of nodes reaches 0.5, i.e., half the nodes are selected. Thus, the performance of the heuristic is almost identical to that of random placement.

The plots in Figure 13(a) show some noise which we attribute to policy routing restricting the paths between client proxies and small size of the initial sets used (20 nodes for the 10 client proxies case).

Based on the experiments performed, we can see that a small number of intermediate overlay nodes, carefully placed, can provide disjoint indirect paths for most of the paths between the client proxies. In the case of 10 client proxies (Figure 13(a)), the number of intermediate nodes required appears to be five. From Figures 13(b) and 13(c) we see that going from five intermediate nodes to 10 and 15 intermediate nodes gives significant improvement but the marginal improvement reduces beyond 15

[4] A better choice of the initial set of nodes could potentially change the performance of the Cluster Heuristic, although the question of how exactly to do this would remain.

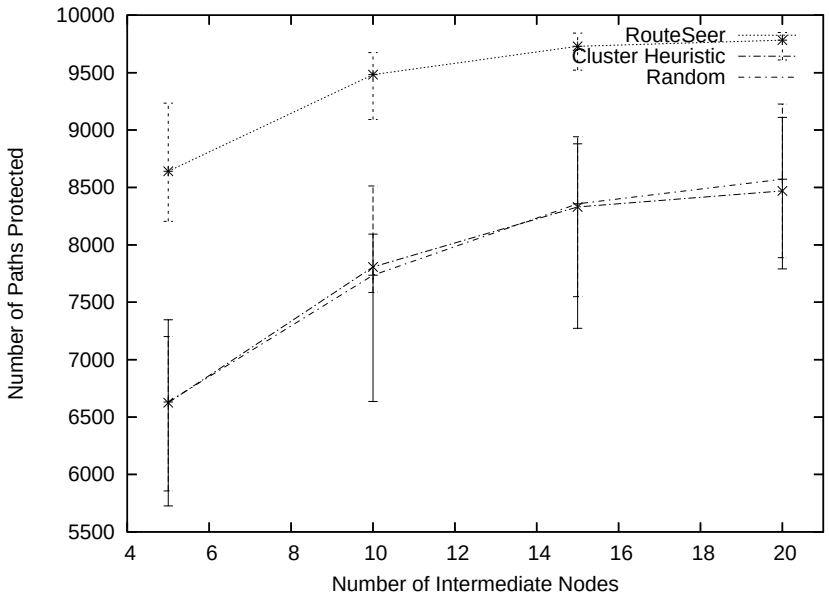


Figure 14: Path Protection with Policy Routing

nodes. Thus, our simulations indicate that for small networks, there are a minimum number of nodes required to provide adequate protection. As overlay size increases, the number of intermediate nodes required as a percentage of the number of client proxies reduces. For example, we observe that approximately 15% intermediate nodes is sufficient for our medium overlays and approximately 10% intermediate nodes is sufficient for our large overlays. In the case of our small overlays, we find five intermediate nodes to be sufficient.

We can draw a similar conclusion by examining the number of paths which are protected using the intermediate nodes. In Figure 14 we plot the number of paths that are protected, i.e., have an indirect path which is disjoint from the direct path between a pair of client proxies, as a function of the number of intermediate nodes used. In terms of the number of paths left unprotected, the reduction using RouteSeer is substantial (e.g., 1259 vs. 3275 for the 5 intermediate nodes case and 118 vs. 1430 for the 20 nodes). We observe that beyond 15 nodes, the improvement in the number of paths protected is small. The graphs for 10 and 500 client proxies are similar.

We also performed some of these experiments without taking into account policy

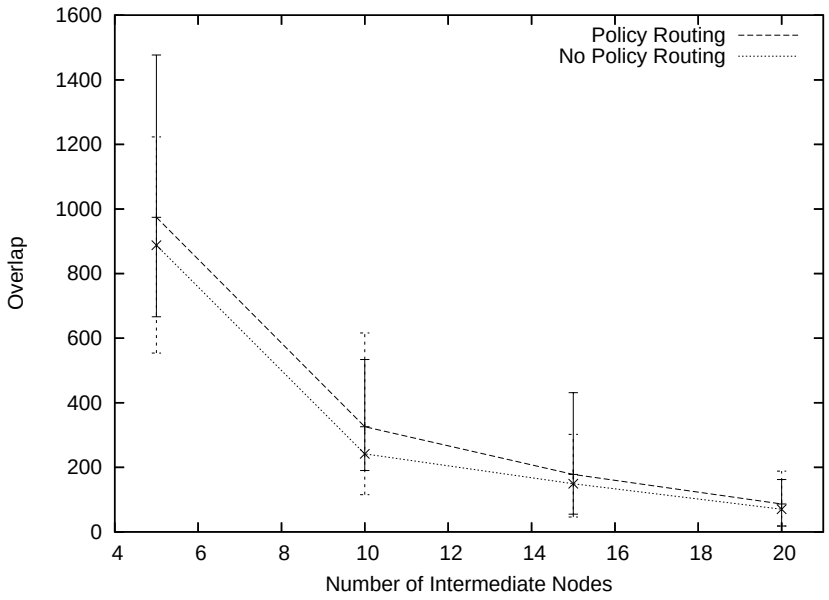


Figure 15: Overlap using 100 Client Proxies with and without Policy Routing

routing for the same topology and with the same seeds for the 100 client proxies experiment as in Figure 13(b). The results of these are plotted in Figure 15. We observe that our simplified policy routing model has slightly increased the total path overlap. This increase is to be expected; the number of possible paths between pairs of nodes is reduced due to the removal of edges from consideration when computing shortest paths using policy routing.

4.4.5 Quality of RouteSeer Solutions

In the previous sections we have seen that the RouteSeer algorithm performs better than the cluster heuristic, but the question remains as to how good is the solution returned by RouteSeer in absolute terms. In other words, how close is the RouteSeer solution to the optimal? Answering this question requires an exhaustive search of all possible solutions which is not practical for the large AS topology. We take two different approaches to answer this question. In the first, we generate synthetic topologies where we can practically run an exhaustive search for the optimal solution and compare the solution generated by RouteSeer to it. In the second, we generate a large

number of random solutions for the AS topologies and compare with the RouteSeer solution.

4.4.5.1 Synthetic Topologies

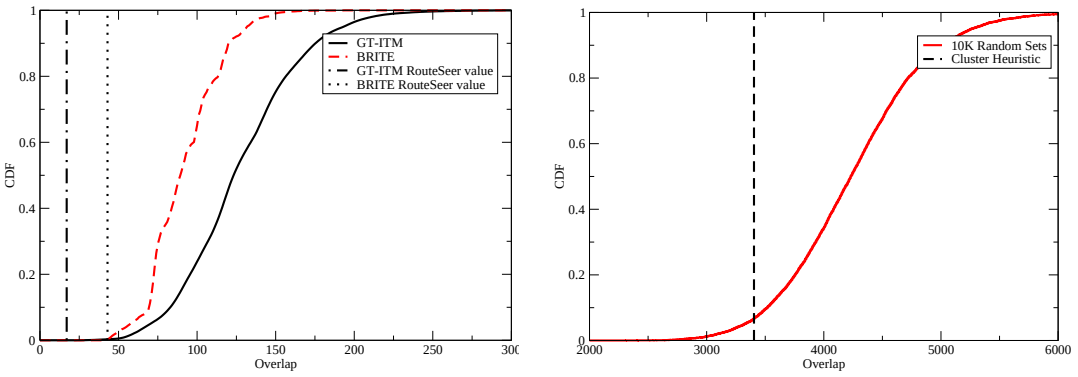
To evaluate the quality of the RouteSeer placements, we use a set of small topologies composed of approximately 250 nodes[5] in which we attempt to place four intermediate nodes for a set of 20 client proxies. We chose this particular set of parameters as it is possible to try all combinations reasonably efficiently to find the optimal set of intermediate nodes in these topologies (the number of combinations is approximately 103 million). To create these topologies, we use two different types of topology generators, GT-ITM [16], a structural generator and BRITE [49], which we use as a degree-based topology generator [79]. For our simulations, we generated several topologies using each generator. For each topology, we computed all possible solutions and compared them with the results produced by RouteSeer.

We plot the results of one representative run of the GT-ITM topology and of the BRITE topology in Figure 16(a). In the figure, we plot the CDF of the possible solutions as a function of the overlap for each set. We also plot two vertical lines, one for the overlap due to RouteSeer placement in the GT-ITM graph (at $x = 17$) and one for the BRITE graph (at $x = 43$). We observe that the placement selected by RouteSeer is very close to the optimal irrespective of the type of topology generator used to create the synthetic topologies.

4.4.5.2 AS topology

For evaluating the RouteSeer algorithm on a realistic AS topology, we use the same topology that was used in Section 4.4.4. We create 10,000 different sets of intermediate nodes of size five and evaluate the overlap of each of these sets. We plot the results as a CDF of the solutions as a function of the overlap in Figure 16(b). The vertical line

[5]The GT-ITM topologies used 245 nodes whereas the BRITE topologies used 250 nodes.



(a) CDF of Overlaps of representative BRITE and GT-ITM graphs

(b) CDF of Overlaps of 10K random sets with 100 client proxies and 5 intermediate nodes

Figure 16: CDFs of Synthetic and AS graphs with RouteSeer

indicates the overlap of the set generated using the Cluster Heuristic. The overlap of the set generated using RouteSeer is 1266 which is beyond the minimum x-axis value implying that the solution RouteSeer generated is significantly better than nearly all the random solutions.

4.4.6 Maximal Disjoint Path Optimality

As discussed in Section 4.2, the RouteSeer algorithm is designed to solve the Maximal Disjoint Path problem. We have seen in our previous experiments that RouteSeer can provide solutions close to the optimal for the Minimum Overlap problem. RouteSeer also provides good solutions to the Maximal Disjoint Path problem but does it provide near optimal solutions to this as well?

To evaluate the absolute performance of RouteSeer on the Maximal Disjoint Path problem, we use the same simulation setup as the previous experiment with the synthetic topologies. We compute the overlap of each set of intermediate nodes as well as the number of paths between client proxies that have non-zero path overlap (minimizing this is the same as finding a solution to the Maximal Disjoint Path problem). In Figure 17 we plot the histogram of the number of solutions on the y-axis that have a particular value of overlap on the x-axis as the solid line. We also plot the histogram of the number of solutions on the y-axis that have a particular

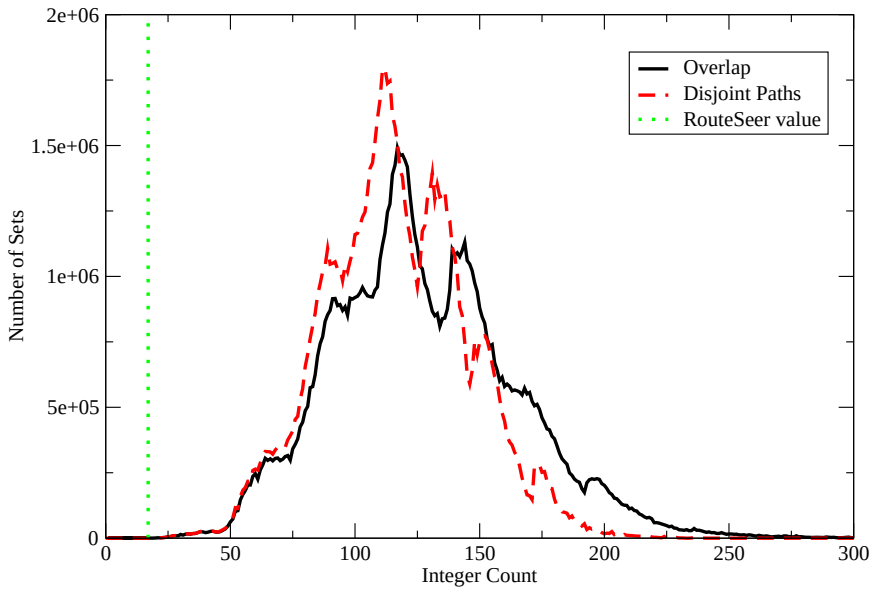


Figure 17: Distribution of Overlaps of all possible sets of 245 nodes

number of paths with non-zero overlap on the x-axis using the dashed line. Finally, we also plot the solution provided by RouteSeer as the vertical line in the figure (at $x = 17$). We see that the two histograms are quite similar and appear to overlap between 0–50. This indicates that good solutions to one problem may yield good solutions for the other as well. The RouteSeer value validates this observation as it provides good solutions to both the Maximal Disjoint Path and Minimum Overlap problems.

4.4.7 Skitter Experiments

Up until now, all our evaluation has been performed using simulations on graphs of the Internet. In this section, we discuss the evaluation of RouteSeer on traceroutes performed on the Internet. The aim of this experiment is the same as for the simulations, namely, to evaluate the performance of RouteSeer in selecting node locations that provide backup paths to the overlay links between client proxies.

We performed a small-scale experiment on the Internet using traces from the skitter project [65]. The data is composed of traceroutes performed from 20 different

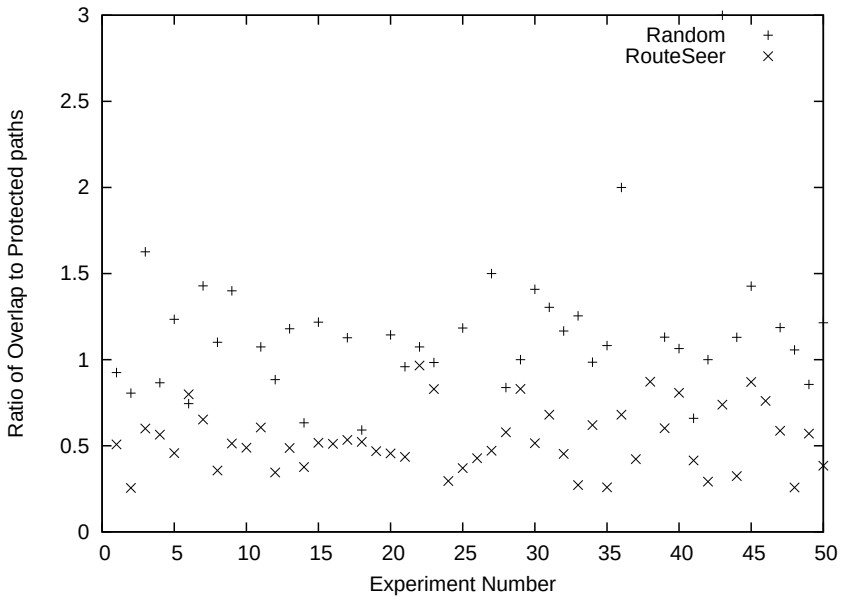


Figure 18: Ratio of Overlap to Protected paths for the skitter locations

locations to a large number of destinations on the Internet (varying from 51K to over 900K destinations). We map the IP addresses in the traceroutes to their corresponding ASes using data on the prefixes advertised by each AS in the RouteViews BGP tables according to the procedure outlined in the skitter project. After performing this mapping and removing repeated ASes, we obtain the AS-level paths from each location to all the destination ASes. We use these AS-level paths to compute the AS routing tables of each source location. We then run the RouteSeer algorithm using the source locations as the client proxies. We use the generated routing tables to select the five overlay node locations which we evaluate using the same skitter traces. For the evaluation, we compute the AS overlap of the direct path between two locations and the indirect path using the overlay node which provides the minimum overlap. We compare the performance of RouteSeer to only Random as the Cluster Heuristic appears to perform very similarly.

In each experiment, we first select a set of 15 locations at random out of the available 20 to act as the client proxies. For the set of 15 client proxies, we compute the intermediate locations according to RouteSeer and also by picking five AS locations

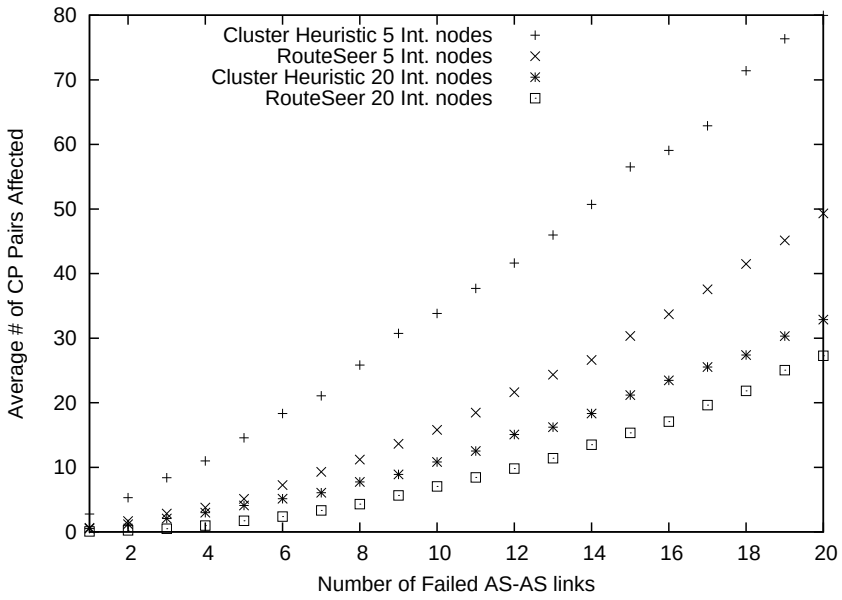


Figure 19: Resiliency during AS-AS link failures

at random. We evaluate the selection and plot the ratio of the overlap to the number of protected paths for overlay locations selected by RouteSeer and those selected at random. This was performed 50 times and the results are plotted in Figure 18. Since many of the traceroutes were incomplete, not all paths were available and hence, we report the number of paths that we tested and the overlap resulting from them rather than the total number of paths. From the figure, we see that RouteSeer performs significantly better than random in most cases. In only one instance was the Random selection slightly better than RouteSeer. This was primarily due to the selection of five locations (out of 15 locations in the experiment for this datapoint) with only a single link to the Internet. Note that these locations violate the assumption that each client proxy have multiple access links. There are also a few instances where the number of protected paths using random placement is zero. This indicates that we could not find a valid traceroute to the that particular set of ASes selected. These sets have no points corresponding to Random placement in the figure (e.g., the 10th).

4.4.8 Overlay Resilience

The overall objective of RouteSeer is to improve the resiliency of overlays by providing backup paths in case of direct path failures. In this section, we investigate how well this goal is achieved in simulation. For this we consider the graph induced by the client proxies and the intermediate nodes, along with the links on the direct and indirect paths between them. In each experiment, we fail a fixed number f of random links and evaluate the resiliency of the overlay in terms of the number of client proxy pairs affected by the failures. We assume that a pair of client proxies is affected if a link on the direct path *and* a link on the indirect path both fail. We repeat this experiment for 1000 iterations using 100 client proxies for each value of f from 1 to 20 for each of the seeds used in Figure 13(b). In Figure 19, we plot the number of failed links on the x-axis and the average number of client proxy pairs affect on the y-axis for different numbers of intermediate nodes. We see that RouteSeer significantly increases the resiliency of the overlays as compared to the Cluster Heuristic when using only five intermediate nodes. The performance using RouteSeer with five nodes is similar to that of using the Cluster Heuristic with 20 intermediate nodes for upto 5 AS-AS link failures. When using 20 intermediate nodes, which inherently provide more path diversity, the improvement is much smaller.

4.4.9 BGP Trace Experiments

The previous experiment with traceroutes shows that RouteSeer performs well when given static route information of client proxies. Similarly, the simulations for overlay resilience indicates that RouteSeer is robust to link failures. To evaluate the performance of RouteSeer in the real world in which routes, rather than single links, fail and are then restored, we devised an experiment using BGP traces gathered from

RouteViews. The RouteViews router peers with and gathers BGP data from 37 different ASes. In our experiment, we consider these 37 peers to be the client proxies[6] comprising the overlay network. We use the BGP RIB data obtained on April 1st 2006 to construct the BGP forwarding tables for each of these overlay nodes. Using these forwarding tables as input, we run the RouteSeer algorithm to place five intermediate nodes to protect the overlay paths.

We use the BGP updates seen by these peer ASes to construct a timeline of the condition of the overlay links. Initially, we consider all overlay links to be "up". If we observe a route withdrawal corresponding to one of the overlay links, we examine the indirect path through the intermediate designated for this overlay link. If the links comprising the indirect path through an intermediate node are still up, we consider that the indirect path can be used to recover from the failure of the direct path. In case either of the links is withdrawn or "down", the direct path is also considered down. The overlay link remains down until a BGP update advertising either the direct path or the links in the indirect path that were down is seen, at which point the overlay link is considered up again. We define the process of an overlay link going down and then coming back up as a *failure* event.

For our experiments, we use the BGP updates seen at the RouteViews router over a three month period from April to June 2006. For comparison, we also randomly select 10 other sets of intermediate nodes and evaluate their performance using the same mechanism. The results are shown in Figure 20 as cumulative number of failure events (on the y-axis) against time (on the x-axis). The duration of unrecoverable failures is important for any overlay service and the shorter this duration the better. To quantify this, we plot the cumulative time that overlay links cannot be recovered as the time columns in the figure.

[6]We actually select the largest prefix advertised by an AS to represent the AS in the processing of the BGP updates.

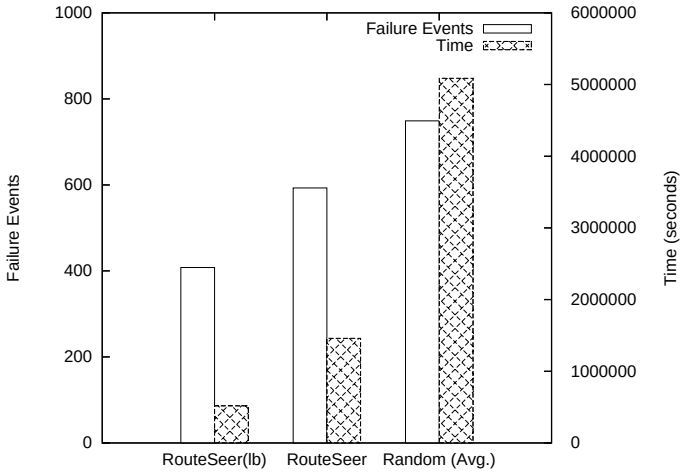


Figure 20: Comparison of resiliency during AS path failures using RouteSeer and Random placement

We observe that using RouteSeer, the overlay network experiences fewer failure events as compared to random placement. Our initial run with RouteSeer selected AS25152 to be the location of the first intermediate node. This is the AS number of the K-Root server [39] and its prefix is advertised from multiple locations on the Internet taking advantage of IP anycasting. For this reason, we exclude RouteSeer from using this AS in all our experiments. The greedy algorithm outlined in Figure 12 results in a skewed allocation of overlay links to intermediate nodes, with more than a thousand overlay links allocated to the intermediate node selected first. For this reason, we also implemented the load balancing discussed in Section 4.3.1.3. The results using load balancing are marked with (lb) in the Figure 20.

It is clear from the figure that load balancing helps in improving the performance of RouteSeer indicating that restricting the number of overlay links protected by any single intermediate node is a desirable. RouteSeer also performs well at reducing the duration of unrecoverable failures. One random placement performs better than RouteSeer with fewer failure events and shorter failure time but overall, the random placements show variation over two orders of magnitude for the failure time. The number of failure events for random placements (mean = 749) has a std. deviation

of 202 indicating that a random choice of intermediate nodes is likely to perform poorly.

4.5 Related Work

We discuss some of the related work in routing and multihoming for path resiliency in this section. Overlay design techniques and optimizations for path resiliency are discussed in Chapter 2.

4.5.1 Routing

Path resiliency has been an important topic of interest for a long time. For example, initially interest was in having multiple routes between source and destination pairs (e.g. [70]). This involved storing multiple routes between source and destination pairs and the solutions usually required modifications to the routing protocols to store more than one route for destinations (e.g. [82]). More recently, the Detour study [67] shows that there exist network paths with better delay and loss characteristics when an intermediate node is used. This idea has been used in overlay networks to create routes in the overlay network that can provide multiple paths between overlay nodes for performance and resiliency (e.g. [7, 21]). Our work differs from these and others in this area in that we are interested in *placing* overlay nodes to provide resilient paths rather than creating resilient routes on the overlay. Our work can be used as the basis for creating an overlay network on which such routing algorithms can be used.

4.5.2 Multi-homing

In a series of two papers [4, 5], Akella et al. analyzed the benefits of multi-homing and compared it to using overlay networks to achieve improved performance and network resilience. In [4] the authors examine the path diversity provided by multi-homed to stub networks. In [5], they compare the performance of multi-homing to that of

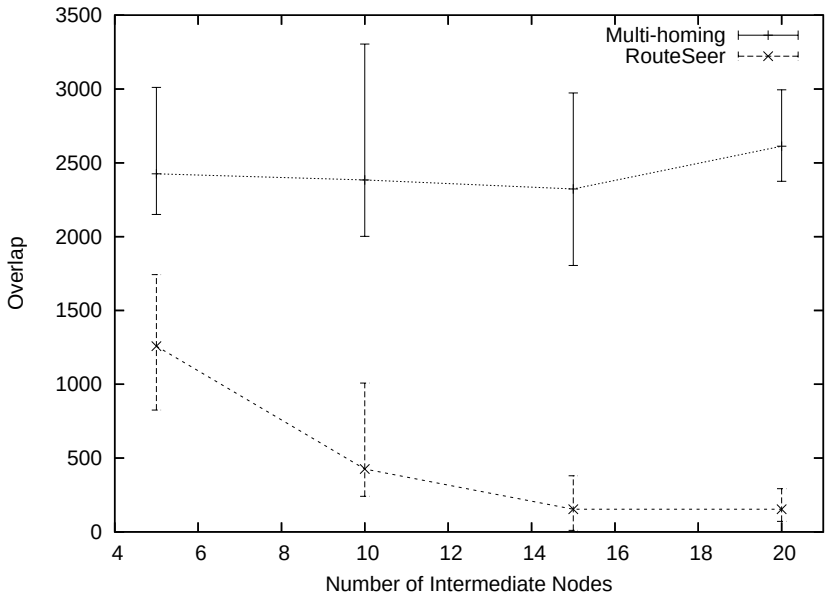


Figure 21: Comparison of multi-homing and overlay placement using RouteSeer

using an overlay network. In particular, they examine the availability of paths using multi-homed stub ASes and compare that to using overlay networks. The authors conclude that if the providers of the stub ASes were chosen carefully, it was possible to achieve performance similar to that of using overlay networks.

An obvious question to ask in light of these papers in the context of RouteSeer is: how much improvement, if any, is there in path diversity using overlay networks designed using RouteSeer instead of multi-homing? To answer this, we performed some simulations using the same setup as in Section 4.4.4. We selected 100 client proxies at random and compute their routing tables using a simplified model of policy routing. After placing the overlay nodes using RouteSeer, we compare the performance of multi-homing to using these overlay nodes in the following manner: we compute the direct path between a pair of client proxies and compare it to the best overlay path and to the best path using one of the providers of the source client proxy. The results of this are plotted in Figure 21. The performance of multi-homing remains relatively unchanged, barring the effect of different seeds, as the number of overlay nodes has no effect on the paths selected. The overlap using RouteSeer decreases as

the number of overlay nodes used increases and remains significantly lower than that of just using multi-homing.

4.6 Summary

In this chapter, we have focused our attention on overlay node placement in infrastructure or service overlays. We studied this problem using overlay network resiliency as the performance objective and proposed an algorithm, RouteSeer, to solve this problem. The RouteSeer algorithm splits the node placement problem into two parts. In the first part, it places nodes called client proxies "close" to the clients of these infrastructure overlays using solutions proposed in the literature. In the second part, it uses the local routing tables available at the client proxies to decide the locations of the intermediate overlay nodes that provide indirect paths which do not overlap with the direct network path between the client proxies. Using only local information ensures that we do not have to depend on possibly incorrect global topology data which may not be available in many instances.

We showed in our experiments that RouteSeer can perform significantly better than existing methods for selecting overlay node locations. Further, our data indicates that for significant overlay resiliency, the number of intermediate overlay nodes required is approximately 10 – 15% of the total nodes in the overlay. We showed that RouteSeer can improve on previous schemes by reducing the number of unprotected paths by a factor of 3–6. the next chapter, we study the performance of RouteSeer on network characteristics such as packet loss and latency.

CHAPTER V

PATH DIVERSITY IN NETWORKS

5.1 Introduction

Routing in the Internet creates a single path between any pair of hosts. Any problem with the available path, such as congestion, routing loops or a failed router, can cause packet losses and poor performance between the pair of hosts. Increasing the number of available paths between pairs of hosts can help mitigate this problem. Proposals to do so include the use of overlay networks [6], multihoming [4], or changes to the BGP protocol to support multiple paths [86, 88] for selected destinations.

Each of these approaches has advantages and disadvantages. Changes to the BGP protocol mainly involve creating and maintaining alternate paths from source to destination in addition to the default. These changes require standardization at the interdomain level, a notoriously slow and difficult process. Multihoming is implemented locally and provides some limited control over alternative paths. Overlay networks can leverage existing links to provide multiple alternate paths to the destination with the potential for greater diversity than multihoming.

Since the three types of proposals use different mechanisms to achieve path diversity, it is difficult to make qualitative comparisons between the various proposals. There has been some previous work on comparing the performance of multihoming and oblivious overlays [5], but that does not indicate how designed service overlays would fare.

In this chapter, we examine the extent of path diversity achievable by various techniques. We start by examining the path diversity of overlays and multihoming in a theoretical framework and make some observations on the path diversity offered.

We show how the proposals that change the BGP protocol can be mapped to one of the techniques offered by overlays or multihoming. We then evaluate the performance of designed overlays using active measurements on the PlanetLab network based on network characteristics such as packet loss and latency. We find that designed overlays are very effective in recovering from packet losses on the direct path. We also observe that in terms of latency, the alternate overlay paths are within a factor of two of the direct paths for 85% of the paths. This performance is significant as the overlays are not specifically designed for optimizing latency. It also justifies the decision to optimize the overlay resiliency as we obtain reasonable latencies on the indirect paths.

The remainder of the chapter is organized as follows: In the next section, we propose the theoretical framework to compare the various path diversity techniques. In Section 5.3, we explain our experimental methodology and discuss the results in Section 5.4. We summarize the chapter in Section 5.5.

5.2 Framework for Overlay and Multihoming Diversity

We use a simple graph model of the AS topology to study the path diversity offered by multihoming and the overlay design proposals. We assume that the nodes in our graph are ASes and we assume that the ASes in which our sources/destinations nodes are located are the overlay nodes. Let $G = (V, E)$ be the graph representing the AS-level topology of the Internet. Paths in this graph correspond to AS-level paths in the Internet. We assume that paths in this graph are constructed using a shortest-path algorithm.[1] In our model, we further assume that paths between nodes are symmetric. Before proceeding further, we highlight a basic property of the trees generated by a shortest-path algorithm.

Observation 5. *Let Figure 22 represent the shortest path tree rooted at source A and assume that equal cost ties are broken consistently. Then, the shortest path from A*

[1]This represents a simplification of the actual policy-based routing used in the Internet.

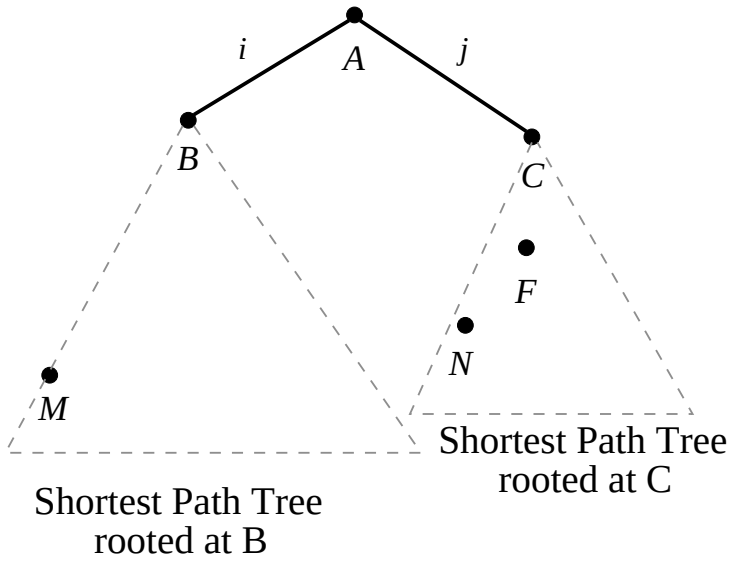


Figure 22: Shortest path tree rooted at node A

to a node M through a neighbor B will not share a common node with any path from A to a node N through a different neighbor C.

Proof. The proof of this observation is quite straightforward. Assume, for the sake of contradiction, that the path from A to M goes through a node F which also lies on the shortest path from A to N. Without loss of generality, let us assume that the shortest path from A to F is through C.

This implies that the path from A to M through F is shorter than the path from A to M through B, indicating that M is in the shortest path rooted at C. This is a contradiction as M is in the shortest path tree rooted at node B. □

This simple observation is the intuition used by the RouteSeer algorithm proposed in the previous chapter. To extend the model to multihomed path diversity, we need to define a few additional terms.

In Section 4.2, the *overlap* between a source S and destination D using an intermediate node I was defined as the number of common network vertices on the direct and indirect paths. We extend the notion of overlap to multihomed paths as follows:

Define $M(S)$ as the set of neighbors of the node S.[2] We define the overlap using multihomed paths

$$mhoverlap(S, D) = \min_{K \in M(S)} overlap(S, D, K).$$

We now formalize the path resiliency using overlays for path diversity between a source S and destination D as

$$resilient_O(S, D) = \begin{cases} 1 & \text{if } \exists I \in O \text{ such that } overlap(S, D, I) = 0, \\ 0 & \text{otherwise,} \end{cases}$$

where O is the set of intermediate nodes. Similarly, we can define the path resiliency using multihoming as

$$resilient_M(S, D) = \begin{cases} 1 & \text{if } mhoverlap(S, D) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

We can now make the following observation on the relative path diversity of using overlays and multihoming.

Observation 6. *For a source S and destination D, $resilient_O(S, D) \geq resilient_M(S, D)$.*

Proof. The proof of this statement is quite straightforward. Clearly, if one of the providers of S provides a disjoint alternate path, the overlay can place an intermediate node in that provider, allowing the overlay to provide the same disjoint path. Therefore, the overlay cannot perform worse than multihoming.

To show that the overlay can perform better in certain situation, we refer to Figure 23. Consider the node S which uses provider M_2 to reach destination D through its provider N_3. The overlay can choose to place the intermediate node I to provide the indirect path $S \rightarrow I \rightarrow D$. For multihoming to provide a similar alternate disjoint

[2]This corresponds to the set of ASes that S could potentially use for transit in the Internet graph though this definition also includes potential clients.

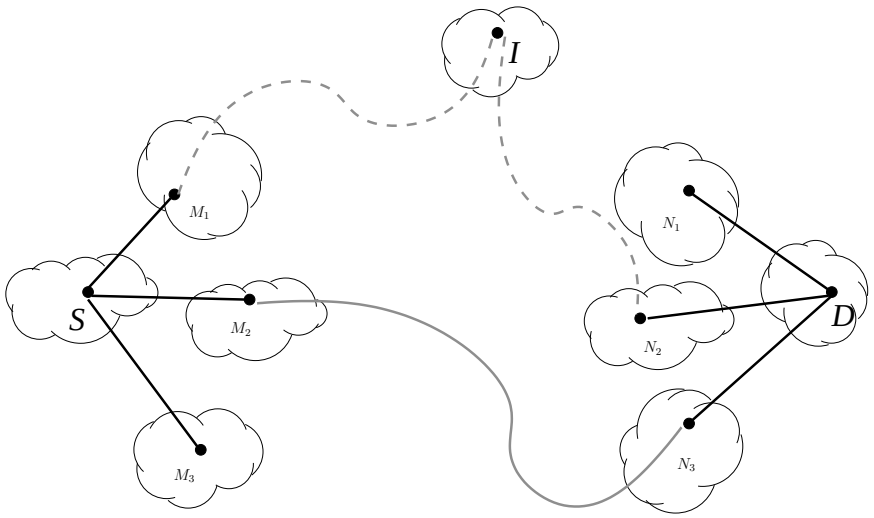


Figure 23: Overlay and multihomed paths between Source and Destination

path, it is necessary that M_1 or M_3 be accessible from D through a provider different from that used to reach M_2. This follows directly from our assumption of symmetric paths and from Observation 5. □

A immediate question that arises is: If one intermediate hop holds the potential of more path resiliency than multihoming, should we use more intermediate hops to increase the potential further? We now show that the answer to that question is most often no, i.e., one intermediate node is sufficient. To show this, consider an indirect path between S and D through three intermediate nodes as in Figure 24. The path from D to I_1 uses N_3 to reach the node and so I_1 does not meet the condition for path resiliency. Once an intermediate node is reached which is on a different outgoing link than the default from the destination D, in this case I_2, we have already achieved the objective of an alternate disjoint path, and so the final path can be $S \rightarrow I_1 \rightarrow I_2 \rightarrow D$. The hop from $S \rightarrow I_1 \rightarrow I_2$ can be eliminated, leaving a single intermediate hop, $S \rightarrow I_2 \rightarrow D$.

Although most cases are covered by the previous example, consider the network in Figure 25. The direct path from source S to D goes through M_2 and N_2. Node I_1 is reachable from S through M_1 but for the destination D, I_1 is only reachable

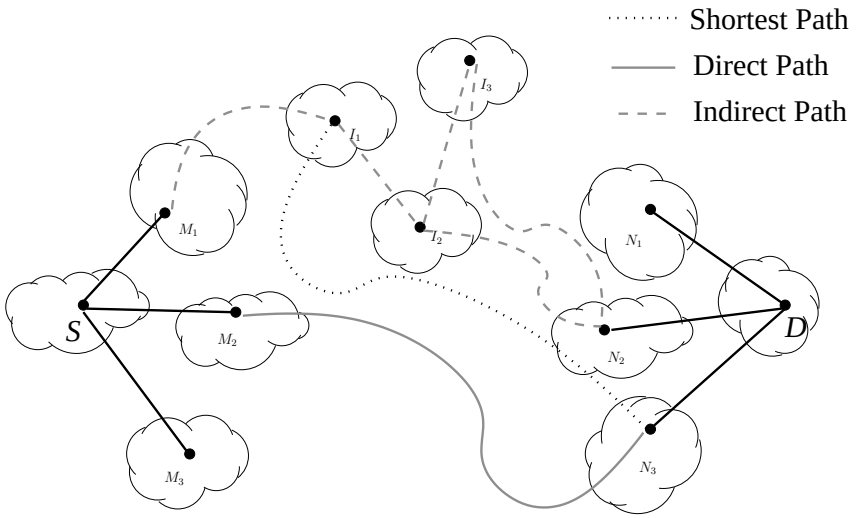


Figure 24: Overlay path between Source and Destination using multiple Intermediate nodes

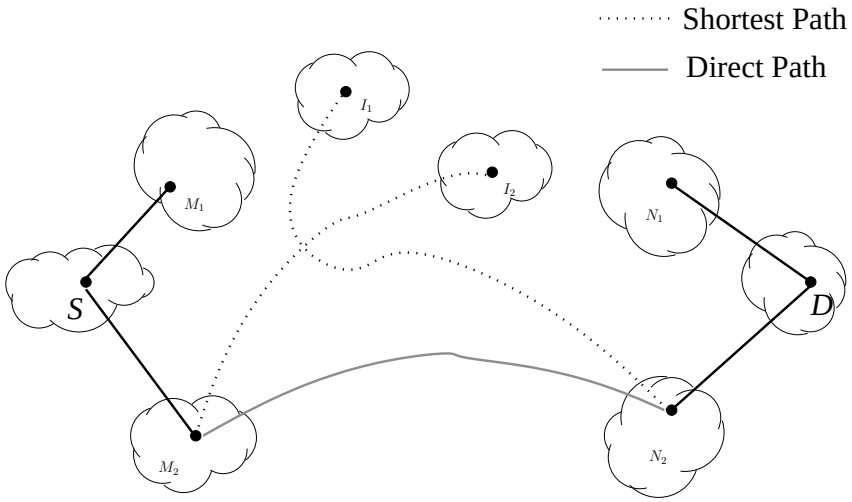


Figure 25: Overlay path between Source and Destination requiring multiple Intermediate nodes

through N_2 and vice versa for I_2. In this case, the set of locations reachable from S through M_1 and from D through N_1 is empty and so, a single intermediate hop is not sufficient to provide path resiliency. The prevalence of this particular case in the Internet is debatable in ASes with outdegrees greater than 2. Thus, in most cases of overlay networks designed for path resiliency, a single intermediate hop should be sufficient. Note that this is the conclusion drawn by many previous empirical studies on overlay networks, although they mostly considered performance metrics rather than path diversity.

Based on the above discussion, we observe that there are two primary mechanisms to obtain path diversity: choose one of several adjacent nodes (other than the default) to route traffic to a destination, or to choose an intermediate node other than the adjacent nodes as a relay for the traffic to the destination. Choosing the first approach of multihoming gives us one point, the source, to decide the path of the traffic to the destination, whereas using an overlay allows us to choose both the link from the source and the link into the destination through the choice of the overlay node.

5.2.1 Other Path Diversity Proposals

Several proposals have recently been put forward for improving path diversity between nodes on the Internet. In a proposal for multi-path interdomain routing (MIRO) [86], the authors extend the BGP protocol to allow for selectively exporting and exchanging additional routes between adjacent ASes. In addition, they also propose tunneling traffic to other (non-adjacent) ASes, after which the packets use regular Internet routes to reach their destination.

Clearly, the two methods of increasing path diversity, namely, export of additional routes and tunneling traffic map directly to choosing an alternate provider and using an intermediate node in our framework. Thus, all observations made would hold for the MIRO architecture as well.

In another proposal [88], Yang and Wetherall propose to allow routers to divert packets from their default route onto other paths, thereby providing alternate paths. This technique is mostly for intra-domain diversity but they also extend these deflections to AS-level decisions. The deflections to adjacent ASes can be seen as using an alternate provider in our framework, except that instead of choosing this alternate AS at the source, it is chosen at differnt points on the path. We can consider the location at which a decision to deflect from the default route is taken, to be the source of a truncated path and our observations about path diversity would hold for the truncated path.

5.3 Experimental Methodology

To evaluate the effectiveness of overlay design proposals and their performance from the point of end users, we would ideally like to evaluate from many nodes in different locations on the Internet. We use hosts from the PlanetLab network to approximate this scenario. Note that most PlanetLab nodes are in academic or research institutions which are usually well connected to Internet. Thus, the connectivity and network locations of the nodes are not completely representative of ordinary end users. [11]

RouteSeer requires access to local routing tables in order to determine the location of the intermediate nodes. Such information for the PlanetLab nodes is not readily available and so we approximate it in the following manner: For each routable AS in the Internet[3], we do a hop-limited traceroute to a prefix in that AS. We then convert the trace into a list of ASes traversed. By parsing this list, we can obtain the next-hop AS for each of the destination ASes in the Internet.

Many routers on the Internet do not reply to traceroute probes. This restricts the effectiveness of the various overlay design proposals as it limits the topology information that can be gathered about the overlay paths. For our experiments, the

[3]We obtain a list of such ASes from RouteViews [61].

traceroute filtering causes many of the routing tables we generate for the PlanetLab nodes to be incomplete. This adversely affects the performance of RouteSeer as those ASes with incomplete entries are excluded from being potential intermediate node locations.

We select a set of PlanetLab nodes to be the overlay nodes of our designed overlay. The direct paths between these nodes are protected using a small set of intermediate nodes selected using the RouteSeer algorithm. Since the total number of paths, both direct and indirect, are quite large we restrict our active probing to a subset of the direct and their corresponding indirect paths. We also place a set of *random intermediate* overlay nodes in order to compare the performance of designed overlays with oblivious overlays.

To compare the performance of this designed overlay to multihoming, we use the following technique from [5] to emulate the behaviour of multihomed sites. To create a multihomed site, we select a set of PlanetLab nodes that are in the same geographic area and which connect to different upstream providers. We replace these nodes by a single *virtual* node that is multihomed to this set of PlanetLab nodes, and consider the path from each of these PlanetLab nodes to a particular destination to be an alternate path for the virtual multihomed node.

Figure 23 illustrates the network topology we use for our measurements. Source M_1 has a direct path $M_1 \rightarrow D$ to destination D. We define the multihomed site for a virtual source S as a set of geographically co-located PlanetLab nodes M_i, each of which is connected to the destination D through the Internet.

We perform active probing using ICMP ping over each of the aforementioned native paths every 30s. Since some networks on the Internet drop port-based ICMP and/or UDP traffic, we use a fall-through mechanism for probing. If ICMP ping fails, we try sending a UDP based probe, and if that probe fails, we use TCP probe on port 22 (ssh) with a ping timeout of one second. A probe is deemed lost if all three

probing mechanisms fail.

In all our experiments, we ignore single probe losses as the loss of a single probe is likely to be a congestion event rather than a failure. Losses can be caused either due to network failure or host (sources M_i, destination D, or intermediates I) failure. By host failures, we imply failures due to access links to the host, and those due to host downtimes. In the PlanetLab network, we notice that access links such as DSL lines usually have a large number of loss events. We identify and remove such host failures by comparing failure events between all measurements to or from the problematic host: a failure across all links at the same time indicates a host failure.

We characterize paths using *recovery rate*, defined as the number of losses on the direct path that were recovered using overlays or multihoming. We also analyze paths for loss event lengths and number of loss events.

In our experiments, we use a diverse set of nodes for the source-destination pairs: we have 33 nodes in the US, 7 nodes in Europe, 3 nodes in Asia, and 1 node in Brazil. We define 21 multihoming sites constructed as follows: we find geographically co-located clusters of nodes (five in number), shown in Table 4. From each of these sites, we construct further multihomed sites by taking all possible subsets of each cluster. We thus have 21 multihomed sites in all. Our experiments are divided into two categories: PlanetLab and Global. The PlanetLab experiments place intermediate nodes within the PlanetLab network. We also add three nodes from the RON [6] testbed for the Global experiments. These experiments are performed on a more global scale, and place the intermediate nodes at hosts other than the PlanetLab or RON nodes. We describe these experiments next.

5.3.1 PlanetLab Experiments

In these experiments, we choose our sources M_i, destinations D, and the intermediate overlay node among PlanetLab nodes. We also define a set of 2- and 3-multihoming

Table 4: Multihoming sites

Site name	Nodes
Boston	lefthand.eecs.harvard.edu planetlab-02.bu.edu planetlab5.csail.mit.edu planetlabtwo.ccs.neu.edu
Bay area	planet2.scs.stanford.edu planetlab2.ucb-dsl.nodes.planet-lab.org sanfrancisco.planetlab.pch.net
Washington	planetlab1.isi.jhu.edu planetlab1.pbs.org plgmu1.ite.gmu.edu
Seattle	planet1.seattle.intel-research.net planetlab03.cs.washington.edu
Pittsburgh	planet3.pittsburgh.intel-research.net planetlab1.cs.pitt.edu

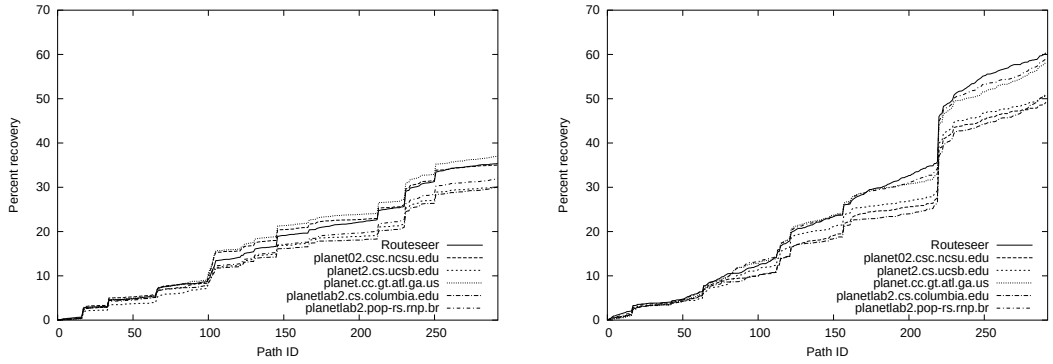
sites, each chosen as a subset of the set of source nodes M_i.

Once we choose the PlanetLab nodes, we use our fall-through ping technique to estimate the downtimes on each of the links in our overlay/multihoming topology. For each source-destination pair (S, D) in our list of paths, we probe the paths $S \rightarrow D$ from source S, $S \rightarrow I$ from source S, and $I \rightarrow D$ from overlay node I, for evaluating RouteSeer's performance. Similarly, we probe the paths $S \rightarrow I_R$ from source S, and $I_R \rightarrow D$ from random intermediate node I_R, for evaluating random overlay placement.

For each multihomed site, we place the multihomed egress links from virtual source S on PlanetLab nodes M_i, $i = 1, 2, 3$. For each multihomed site, we probe the paths $M_i \rightarrow D$ from M_i.

5.3.2 Global Experiments

In these experiments, we choose our sources S, and destinations D from the set of PlanetLab and RON nodes. The RON nodes are among commercial networks and provide more diversity for our experiments. We also ensure that our source and destination nodes are globally distributed. Our intermediate nodes are placed outside



(a) With GTI-DSL (b) Without GTI-DSL

Figure 26: Loss recovery comparing RouteSeer to random intermediate nodes

the PlanetLab and RON networks.

Similar to the PlanetLab experiments, we use our fall-through technique to probe the paths $S \rightarrow D$ from source S, $S \rightarrow I$ and $S \rightarrow I_R$ from source S. However, since we do not have access to overlay nodes I, we probe the paths $I \rightarrow D$ and $I_R \rightarrow D$ from the destination D, under the assumption that our paths are symmetric.

5.4 Results

We outline the results of our experiments on the PlanetLab network performed between the 25th and 30th of November, 2006.

5.4.1 Recovery from losses

We begin by plotting the number of losses recovered by the RouteSeer intermediate node as well as the random intermediate nodes. In Figure 26(a), for each path on the x-axis, we plot the cumulative fraction of losses recovered up to that particular path on the y-axis. Out of the total of 397 direct paths probed, 93 paths did not show any losses on the direct path and are removed from further consideration. Out of the 304 paths remaining, we observe that the performance of the RouteSeer and random nodes split into two groups. The group composed of the RouteSeer node, along with the `ncsu.edu` and `gt.ga.us` nodes performs better than the other nodes. Within the

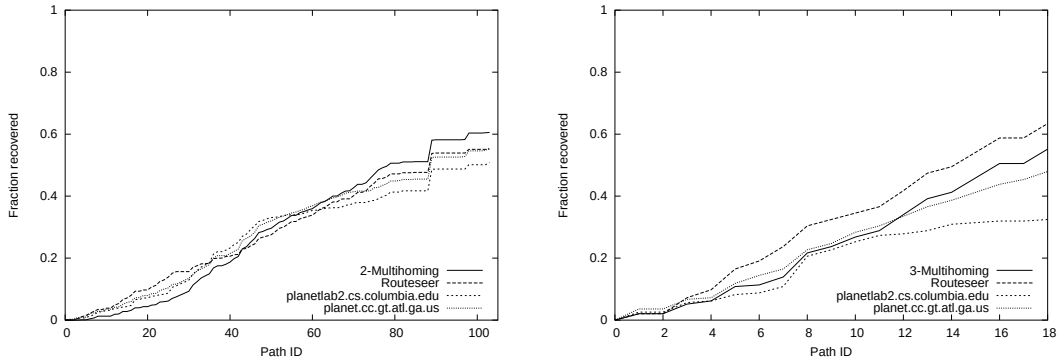
group with the RouteSeer node, the `gt.ga.us` node performs the best.

To investigate the cause for the surprising performance of the `gt.ga.us` node, we examined the pattern of losses and recoveries of each of the source overlay nodes. One of the overlay nodes, planetlab1.gti-dsl.nodes.planet-lab.org (GTI-DSL) had very large numbers of losses as compared to the remaining nodes. The 12 paths in which the GTI-DSL node is featured had on average 821 losses, while all 292 other paths had only 16 losses on average. Considering the GTI-DSL node as an outlier, if we remove the paths featuring this node, the resulting plot in Figure 26(b) shows that the RouteSeer node now performs better than all the other random nodes. Since we have removed a large number of unrecovered losses, the percentage of recovered losses also increases substantially. We also observe that the two clusters of nodes exists in this case as well, but the random nodes that now perform well are `gt.ga.us` and `rnp.br`.

From the two figures, we observe that the performance of the random nodes is variable and depends on the specific paths selected. Changing the set of paths changes the order of the random nodes significantly. For example, the `ncsu.edu` node is in the better cluster in Figure 26(a) but is in the worse cluster in Figure 26(b). On the other hand, the performance of RouteSeer is very stable in that it is either the best or close to it in both cases. The `gt.ga.us` node also remains in the better cluster but it is not clear how it would be selected a priori over any other random node.

5.4.2 Performance relative to multihoming

We now compare the performance of the RouteSeer intermediate node to multihoming using our 21 multihomed sites. For the multihomed sites, we designate one of the paths to the destination to be the primary path. If there is a loss on the primary path, we switch to an alternate multihomed path in an effort to recover from this loss. This is an idealized scenario with perfect information of all the losses and the



(a) Comparing RouteSeer to 2-multihoming (b) Comparing RouteSeer to 3-multihoming

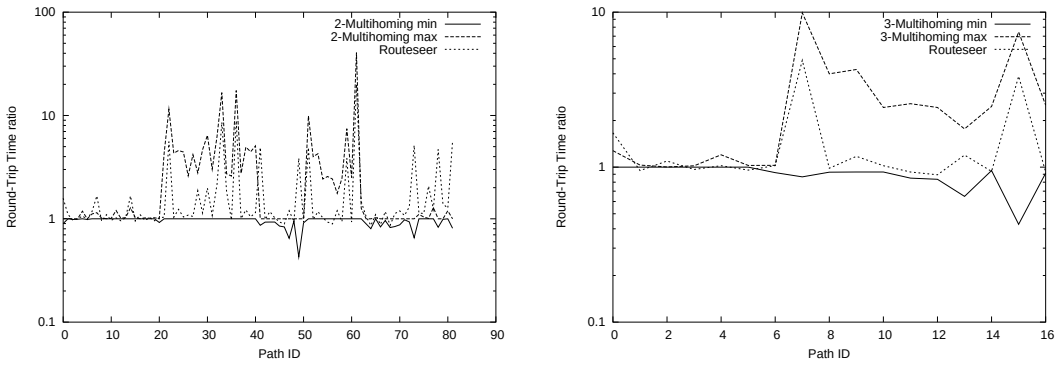
Figure 27: Loss recovery comparing RouteSeer to Multihoming

paths to switch to. Since we are only interested in the best performance achievable using this technique, we ignore the complex issues of path selection and switching.

We plot the cumulative fraction of losses recovered for 2-multihoming and the RouteSeer intermediate node in Figure 27(a) and for 3-multihoming in Figure 27(b). We also plot two random nodes, one from each cluster observed in Figure 26(b) for comparison. We observe that for the 2-multihoming case, the RouteSeer node performs well for the initial set of paths, but eventually the multihoming of the source nodes allows them to recover from more losses than the overlays. In the case of 3-multihoming, the results are reversed with 3-multihoming performing worse than the RouteSeer intermediate node. This is surprising since 3-multihoming has two alternate paths available to choose from as compared to the one alternate path in all the overlays. Note that the number of paths available in this case is much smaller because of experimental constraints. The small sample size together with the small number of losses, an average of 11 losses per path, causes this result.

5.4.3 Latency performance

In addition to loss recovery, we also compare the performance of the overlay against multihoming based on the end-to-end latency through the direct and indirect paths. For this experiment, we compute the latency of the indirect path as the sum of the



(a) RouteSeer and 2-multihoming (b) RouteSeer and 3-multihoming

Figure 28: Latency performance of RouteSeer and Multihoming

average latencies from source to intermediate node and from intermediate node to destination. For the multihomed paths, the latency through the alternate path is computed in two ways, as the maximum of the mean latencies of the multihomed paths used and the minimum of the latencies of the paths used. We plot both values to provide a range that multihoming can achieve, depending on the paths that are chosen.

We plot the latency performance relative to 2-multihoming in Figure 28(b) and to 3-multihoming in Figure 28(b). For each path, we plot the ratio of the latency of the indirect or alternate path to the latency of the direct path. On the y-axis, we plot the ratios using a log scale while we plot the path indices on the x-axis. From the figures, we observe that the performance of RouteSeer is similar to that of the maximum latencies that multihoming achieves *despite* the fact that the RouteSeer algorithm does not optimize for latency. In fact, RouteSeer performs well within the envelope of the maximum that 2-multihoming achieves in most paths except for pathIDs greater than 70. In case of 3-multihoming, RouteSeer always lies within the envelope of the maximum multihoming latencies.

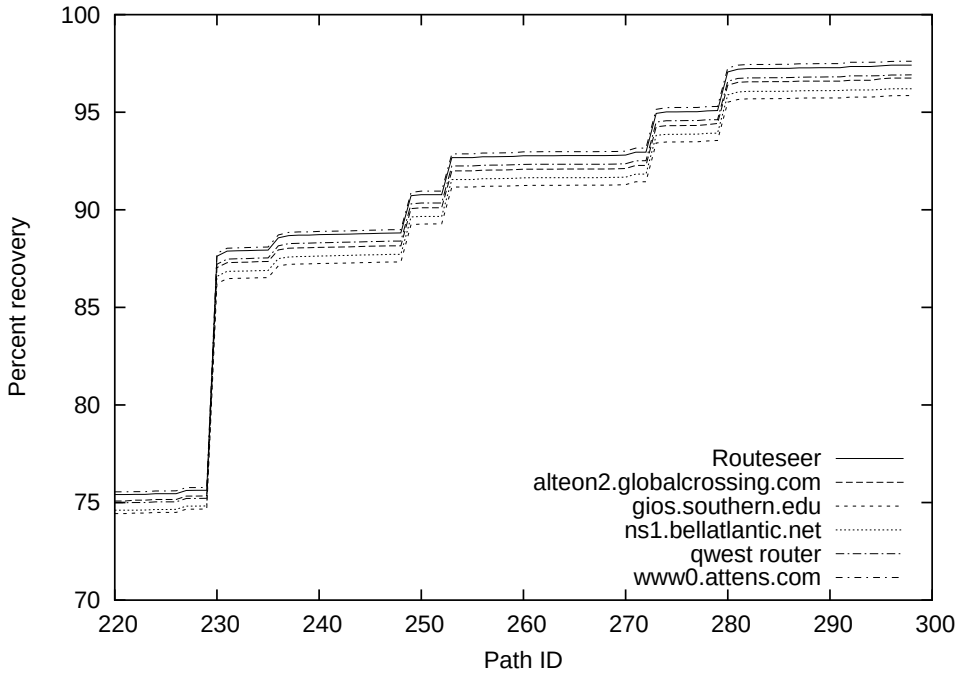


Figure 29: Loss recovery comparing RouteSeer to random intermediate nodes for the Global experiments

5.4.4 Global Experiments

We now discuss some of the results from the Global experiments. The intermediate nodes selected by RouteSeer for these experiments are outside the RON and PlanetLab networks. Specifically, they are located in the Sprint and Cogentco ASes which are tier-1 ASes. Instead of choosing the other intermediate nodes in a random fashion, we selected them to also be in other tier-1 ASes. The specific ASes we selected were Verizon, Global Crossing, Qwest and ATT.

We plot the recovery rate in Figure 29 for all the intermediate nodes. We plot the path IDs from 220 onwards as the performance of the nodes was very similar for the earlier paths. We observe that RouteSeer, just as in the PlanetLab experiments, performs very close to the best. The best intermediate node turns out to be the ATT node with a 97.6% recovery rate while RouteSeer has a recovery rate of 97.4%. Thus, simply choosing the intermediate node to be in a tier-1 AS is not guaranteed to give

the best results, but RouteSeer consistently provides extremely good performance.

5.5 Summary

In this chapter, we proposed a theoretical framework to compare the path diversity provided by different proposals, namely, changes to the BGP protocol, multihoming and overlay networks. We design an overlay network using RouteSeer and evaluate its performance using active measurements of user metrics such as packet loss and latency. We show that the RouteSeer-designed overlays perform as good or better than the best random selection of intermediate nodes and can recover almost 98% of outages on the direct path. We also show that this recovery can be achieved while keeping the latency on the backup paths to within a factor of two of the latency on the direct paths.

CHAPTER VI

LUNA: DESIGN OF A SERVICE OVERLAY

6.1 Introduction

As a growing number of applications are being implemented over the Internet, the requirement for a fast, reliable and accurate naming service is increasingly being felt. Applications such as Internet telephony require a naming service to provide a mapping between a user identifier and its record. Other applications such as Presence require a resource lookup service to map a resource to a network location and to update this mapping frequently. Many of these applications expect operations such as record lookup and updates to user records to complete in a bounded time period.

Initially, services such as naming and resource lookups were provided using a client-server architecture, with clients contacting a designated server to perform the lookup. This has evolved into distributed architectures which can be hierarchical, such as the DNS hierarchy or more recently as decentralized peer-to-peer networks. In the DNS hierarchy, each client is provided a designated DNS server to contact. This server resolves names on behalf of the client by looking into its database for the particular name. In case the server does not find the name, it then queries another server which has a wider view of the namespace. This is possible due to the hierarchical nature of the names used in DNS.

Recently, structured and unstructured peer-to-peer networks have been proposed as alternatives for creating a distributed lookup service. In a structured peer-to-peer network, a query from a client (which is usually a peer in the network), is routed through one or more hops to the peer whose address space contains the query. The number of hops can be bounded either by a horizon as in the case of unstructured

peer-to-peer networks or by a function of the number of nodes in a structured peer-to-peer network. Regardless of the nature of the peer-to-peer network, these protocols assume that there is high churn in the nodes composing the network. Due to this assumption, these networks cannot directly provide a highly reliable service.

The naming service for applications such as Internet mobile telephony requires maintaining records for millions of users and services that are required to be available whenever a call is set up or a service is requested from the network. The user records also need to be constantly updated with network location and usage details as the users move in the network and use network services. These two requirements of time-bounded lookups and high update rates impose a set of unique requirements on the naming service.

This chapter describes the design and operation of a distributed naming service with the specific goals of being highly scalable, with low and bounded query and update latency, with low maintenance requirements and also provide highly accurate query responses. We also compare other approaches to constructing such a service and explore the tradeoffs involved in the design choices we make. To achieve these goals, we construct a managed service overlay based on a structured one-hop peer-to-peer network [33]. We assume that the nodes in this network have a low failure rate allowing for a low overhead management protocol. We use data replication to provide fast recovery after node failures, and in conjunction with limited caching, provide improved query response. The limited amount of data replication allows for very low update latencies.

The rest of the chapter is organized as follows: In the next section, we describe the design goals in detail and compare different approaches to achieving these goals. In Section 6.3, we describe our service overlay which is called LUNA. We discuss extensions to our service that can provide multi-attribute queries in a scalable manner based on the Mercury [13] peer-to-peer network in Section 6.4. We evaluate the design

of LUNA using extensive simulations in Section 6.5 and summarize in Section 6.6.

6.2 Design Requirements

As outlined in the introduction, a generic naming or resource location service for applications such as Voice over IP (VoIP) and presence requires a mechanism to store and quickly retrieve the records of millions of users and services. These user records can change dynamically as the user moves in the network, neccesitating a robust update mechanism to handle these user record updates. Based on this application scenario, we can list the following requirements that a flexible resource location service should satisfy:

- **Fast and Time-bounded query resolution:** At its core, this is the primary function of a resource location service, to answer queries. We require queries to be responded to quickly within a fixed time bound, at least for the primary attribute in a multi-attribute service. As an example of a time bound, VoIP and presence applications require queries to be answered rapidly ($\tilde{1}$00s milliseconds) as there are multiple queries and packet exchanges for a single call setup or message transfer.

- **Reliable:** Since most transactions in these applications begin with a resource or name lookup, the naming service has to provide responses to queries with high reliability. By this, we mean that the service should attempt to answer the queries in spite of network failures and node failures.

- **Updates:** As users move in the real world, their network addresses and geographic information are likely to change . These changes must be reflected in the corresponding records stored in the resource location service as quickly as possible to avoid giving responses with stale information. These updates restrict the types of caching strategies that can be employed by the naming service.

- **Multiple attribute searches:** Given user records with multiple attributes, complex queries such as range queries can also be generated that search on more than one attribute. This imposes stricter requirements in terms of the arrangement of the user records, e.g., hashing of user ids to store the records is no longer possible [13].

These requirements together impose certain constraints on the design of the naming service. We outline some of the restrictions as we evaluate current approaches in terms of the requirements outlined above.

6.2.1 Current Approaches

Current designs for such a resource location service include the Domain Name Service (DNS) and unstructured and structured peer-to-peer networks. A VoIP application, Skype [66], uses an unstructured peer-to-peer network based on KaZaa as its naming service while structured peer-to-peer networks have been proposed as a substitute for both DNS [62] and Skype [71].

The DNS hierarchy is an example of a scalable resource discovery mechanism which maps domain names to IP addresses. The records stored in the DNS hierarchy are relatively static and the design of the DNS hierarchy takes advantage of this to cache the query results, speeding up subsequent responses to queries. This caching (for periods upto a day) fails in the face of dynamic data changes, and incorporating a mechanism to invalidate cache entries would not be practical. Furthermore, the query response times can be up to five seconds in case of timeouts [43].

Unstructured peer-to-peer networks such as Gnutella [26] and KaZaa [38] are instances of resource location mechanisms. These networks flood a request for a particular key (in this case a search term) up to a defined number of hops. If a node within this horizon has a resource (typically, a file which matches the search term), it responds to the querying node. There are several problems with this approach:

- **False negatives:** The number of hops a query can traverse is usually bounded. In case the record being searched for is beyond this horizon, the querying node will not get a response.

- **Unbounded query response time:** There is no limit to the time in which a response must arrive. The time taken to decide if a query failed would depend on the number of nodes in the horizon and the query propagation delay between nodes. Typically, it has been observed that queries can take upto several seconds to generate responses.

- **Reliability:** These networks are usually best-effort. The nodes in these networks are usually composed of the clients of the application. These clients can join and leave the network at will, causing queries to be dropped.

There are also concerns about the scalability of these networks when the number of users increases into the 10's of millions [19]. Caching can be used to improve the query response time once a query has been answered, but the dynamic nature of the user records could lead to the caches serving out stale information. A mechanism to invalidate the caches whenever a record changes would overwhelm the network with this traffic.

Structured peer-to-peer networks have been proposed to mitigate the problems of unstructured peer-to-peer networks described above. A typical structured peer-to-peer network operates in the following manner: A large keyspace is used in which each record that is to be stored is hashed to a unique key in the keyspace. Each node in the network is assigned a portion of the keyspace, i.e., it is responsible for storing the records that lie in that portion of the keyspace. Each node creates links to a small number of neighbors which are selected in a structured manner based on their keys. When a node searches for a key, it forwards the query to the neighbor whose key is most similar to the query. This continues until the query reaches the node

responsible for the searches key, which either responds with the record or returns an error. This basic mechanism has been improved in several different ways. Most structured peer-to-peer networks use some form of ID hashing to ensure that the records are uniformly distributed over all the nodes in the network which precludes efficient range queries. Hence, these networks cannot be directly used for the naming service that we are require.

None of the techniques described above meet all the requirements that we envision in a reliable, efficient naming service, but many of the properties are present in the current proposals. We base the design of our resource location service, called LUNA, on several features from these proposals. We next describe the design of LUNA in detail.

6.3 Design of LUNA

There have been recent designs for structured peer-to-peer networks which use only one or two hops to reach the node storing the user record. We use one such design by Gupta et al. [33] as the basis of our overlay network, but with the important distinction that we do not used hashed identifiers for the records. We also use the idea proposed in Mercury to handle queries in multiple attributes by using different overlays of the same set of nodes for each of the attributes that we want to search on.

LUNA is designed in the form of multiple overlays, one for each attribute that the service stores. This approach is not scalable to an arbitrary number of attributes but is sufficient when the number of attributes is small (less than 10). We designate one attribute as the primary attribute and require it to be unique for each record stored in LUNA. In this chapter, we restrict our focus on making the lookup of the primary attribute be $O(1)$ and providing very high reliability in answering these queries. In the remainder of this section, we describe the working and construction of the overlay for the primary attribute, called the primary overlay. We explain the extension to

multiple attributes in Section 6.4.

We now define some of the terminology that we use in the rest of this paper. We use *nodes* to refer to the machines that constitute the LUNA service. These nodes store user information including the different attributes. *Users* are entities that store and retrieve records from the nodes using registration messages and queries.

We begin by stating some assumptions on which the LUNA service is based. We assume that this naming service is run on dedicated nodes, and so the nodes that compose this service are assumed to be long-lived, i.e., they remain on the network for long periods of time and the churn in the overlay network is very low. The users who use this service can be mobile and their time logged into the system can be variable. We also assume that each user has an identifier of the form <user-id>@domain where the user-id is unique in the domain. Note that this identifier has the same format as the SIP URI.

6.3.1 User Behavior

A user that participates in this service begins by connecting to one of the nodes in the LUNA service. We assume that the user has prior knowledge of a set of nodes that it can contact, either by querying a website to get node addresses or by a pre-configured list of nodes available in the software. Once the user is connected to a node, called the user's *proxy*, the user registers[1] itself in the LUNA service by issuing an REGISTER request with its user-id and other optional information. This message is routed to the node, called the user's *home* node, that stores the user's record. This home node makes note of the user's presence and adds the current user location and the location of the proxy into the user record. This user record is also updated at all replicas and caches of this home node. This user can then generate queries for other users in the

[1]We do not consider the problem of authenticating the user to the service in this document. We assume that this can be accomplished by a mechanism such as a secret key which is stored in the user's record and retrieved during the join.

system. These queries are routed to one of the nodes maintaining the target user's record, which then responds with the requested information. Finally, the user can logout of the system using a LOGOUT message that is routed in a similar fashion to the REGISTER request.

6.3.2 LUNA Overlay

The LUNA service is composed of the nodes arranged to form an $O(1)$ overlay described by Gupta et al [33]. The address space of the overlay is the set of all possible user identifiers. Each node in the service is responsible for an address range in the address space. This address range encompasses a set of user-ids and the node acts as the home node for these users. For example, if the address range assigned to the node is [a-f]*, then this node is the home node for user-ids aa, aaaa, b, foobar, and so on. Each node also has knowledge of the address ranges stored on all other nodes in the overlay. Thus, to route a query for any user-id, a node simply looks up the node responsible for the particular address range and forwards the query to it in a single hop. To answer range queries, i.e., queries of the form "user-ids matching foo*", the node forwards the query to all the nodes whose address ranges match the query. Curently, LUNA can answer only queries of the form "foo*" where the prefix is fixed, but with a suitable choice of indices in one of the other attributes, other types of range queries could be answered.

Each node's information is also replicated at several other nodes in the network. When a node a makes a query to node c which is down and the query times out, it then contacts the nodes which replicate c's data in turn to satisfy the query. Thus, a query will get a response from the overlay, barring the failure of a node and all its designated replicas.

In addition to the replicas, each node also has a designated set of nodes that act as caches for the node's information. The primary difference between a cache and a

replica is that a replica has all the information whereas a cache only stores the user records that have been requested from it. Initially, a cache does not maintian any records. When a proxy node contacts the cache with a query for a user record, the cache checks if the user record is already in the cache. If it is, the query is answered, but if the record is not in the cache, the cache then fetches the user record frrom the home node, inserts the record into its cache and responds to the query.

Each node in the LUNA service also maintains a set of virtual coordinates based on Vivaldi [22]. The information for this is piggy-backed on the messages that are exchanged between the nodes. Using these coordinates, several optimizations can be made to the basic LUNA operation described above. Initially, when a user is connecting to its proxy node by selecting a node from an available set, it can choose to connect to the node which is closest to it in the virtual coordinate space. Using these coordinates, the replicas and caches for a node's data are selected to be widely spread across the underlying network. This ensures that a network outage in the underlying network in one region would not affect access to all replicas of a node's data. Also, when responding to a query, a node can elect to contact a close replica instead of the actual node to reduce the latency of the response.

6.3.3 Overlay Maintenance

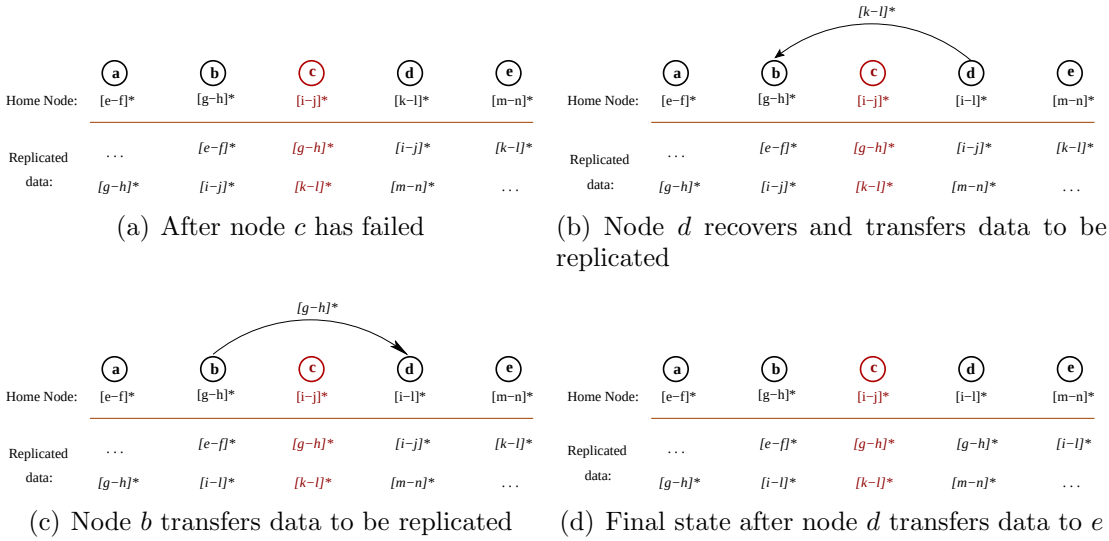
For the routing of queries in LUNA, all nodes have complete knowledge of all other nodes in the overlay. This requires that all nodes be informed of any changes in the address space assignment that take place in the overlay. These events occur in the overlay due to nodes that join the service, nodes that fail or leave the the service and changes in the address ranges that nodes are responsible for. To propagate these changes in an efficient manner, we use the technique proposed by the authors in [33] and impose a hierarchy on the nodes in the system. State information about the nodes is aggregated and sent using this hierarchy. A set of nodes which together advertise a

contiguous address range are aggregated into a *region*. The node responsible for the start of the region user-id range in each region is designated as the *region leader*.

We define the function *pred* on a node n to return the node responsible for the address range immediately preceeding the address range of node n. The address space is assumed to be circular, in that if n is responsible for the beginning of the address range, *pred* returns the node responsible for the last address range. The *succ* function is defined in a similar way to return the node responsible for the address range immediately succeeding the address range of node n. We use these functions to impose an ordering on the nodes and to obtain the predecessors and successors of any node in the system.

Each node maintains a set of k predecessors and k successors that it exchanges keep-alive messages with, where k is a system parameter. We use a value of $k = 1$ and explore the effect of changing this parameter in our evaluation. The predecessors and successors also replicate the data stored on the node. This replication is used to efficiently recover from node failures as explained in Section 6.3.4. If the immediate predecessor or successor of a node fails to respond to three consecutive keep-alive messages, the node will communicate this information to its region leader. If the region leader receives notification from the predecessor and successor of a particular node that it is not responding, the region leader declares the node down. The region leader waits for the node performing the recovery to complete the recovery process. Once the recovery process is complete, the region leader then sends this notification to all other region leaders. Any other hierarchy messages received in this period are aggregated and sent together. This message is then disseminated by the region leaders to the members of their region.

We make some observations on this design decision of using a specific hierarchy to disseminate the overlay updates. The authors in [33] have described some of the advantages of using a hierarchy for this purpose. One specific aim of the LUNA service



(a) After node c has failed

(b) Node d recovers and transfers data to be replicated

(c) Node b transfers data to be replicated

(d) Final state after node d transfers data to e

Figure 30: LUNA failure recovery

is to achieve high reliability. Although a hierarchy implies that nodes higher in the hierarchy have more importance, the choice of a hierarchy is not contradictory to achieving high reliability. Since the hierarchy is only used to disseminate the updates and does not affect the resolving of queries, the retransmission mechanism ensures that few queries are lost. The failure of nodes in the hierarchy may cause some node events might be missed but the replication of the data ensures that even if the query initially fails to reach the correct node, it can be retried on a replica. The net effect of this is just a temporary increase in the response time of the queries to the portion of the address space that is affected by the missed updates.

6.3.4 Overlay Dynamics

Although LUNA assumes that the nodes that participate in the overlay are long-lived, nodes can fail or leave. Nodes can also be added to the overlay to increase the capacity of the service. To accomodate the dynamics, we describe the procedures for dealing with node arrivals and departures.

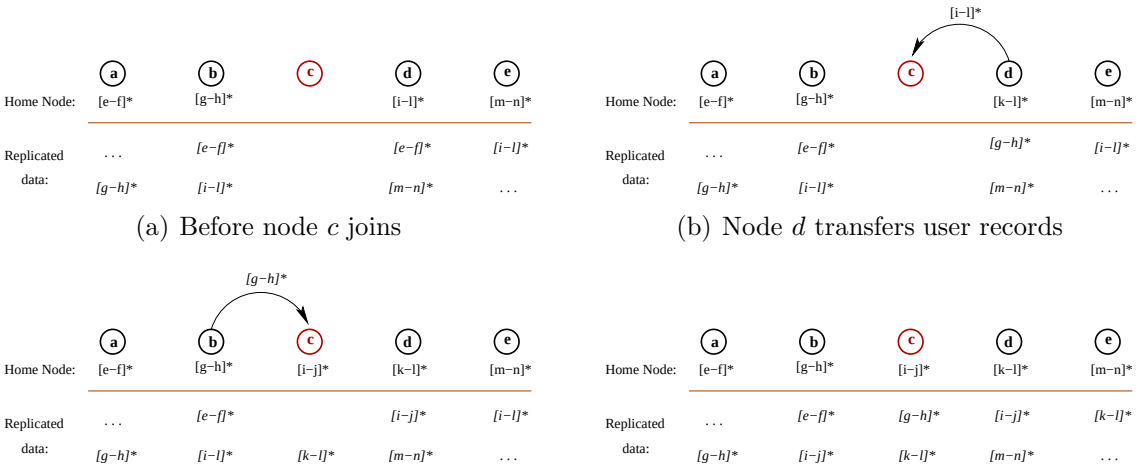
6.3.4.1 Node Failure

We first describe the procedure to deal with node failures or departures. Recall that the content on each node is replicated at its predecessor and successor. When a node fails, the neighbour with the smaller number of user-ids takes over the departed node's address space. In case of a tie, the predecessor takes over the address space. To determine the node with the smaller user-id space, once the failure has been detected, the predecessor sends a keep alive to the successor and vice versa. Once the node with the smaller id-space is established, it runs the recovery procedure outlined in Figure 30. The recovery procedure consists of exchanging data between nodes to maintain the data replication between predecessors and successors. In Figure 30(a), node d is the node performing the recovery. It begins by copying over its data ([k-l]*) to node b which is its new predecessor (Figure 30(b)). The next step is for node b to copy over its data ([g-h]*) to d which is its new successor. At this point, the replication between nodes b and d is complete, but node e does not replicate all of d's expanded address space. For this, d transfers [i-j]* to e, completing the failure recovery.

The process outlined above is for the case when $k = 1$, i.e., there is only one predecessor and one successor that replicates the data. Before outlining the general procedure for $k \geq 1$, we need a couple of definitions. Let $p(n, i)$ be the node reached by performing the *pred* operation i times from location n. Similarly, let $s(n, i)$ be the node reached by performing the *succ* operation i times.

Let c be the location of the failed node. Then, if the node performing the recovery is the successor of c, the recovery algorithm can be written as:

```
for i = 1 to k
do
  Copy user records from p(c,i) to s(p(c,i),k+1)
  Copy user records from s(p(c,i),k+1) to p(c,i)
done
```



(a) Before node c joins

(b) Node d transfers user records

(c) Node b transfers data to be replicated

(d) Final state after node d adjusts e's state

Figure 31: Node join in LUNA

```
Copy user records from s(c,1) to s(c,k+1)
```

It is simple to verify that this algorithm works for all $k \geq 1$. When the recovering node is the predecessor of the failed node, the algorithm is similar except that the $s(.,.)$ and $p(.,.)$ operations are swapped. In case a region leader fails, the node's successor takes over as the leader.

6.3.4.2 Node Join

When a node joins the overlay, it first selects a user-id to decide the portion of the address space to join in. This user-id is only used to locate the region of the overlay to join and can be randomly generated, or more often can be assigned by an operator or during a load balancing operation (see Section 6.3.5). The new node contacts an existing node on the overlay to find the node responsible for the address range in which the user-id lies. The new node then joins the overlay adjacent to the responding node. The address space of the responding node is split and the new node takes responsibility for half the users in that address space. Note that since the distribution of user-ids can be non-uniform, the address space itself may not be split in half.

We illustrate the join procedure using Figure 31 for $k = 1$. In the figure, node c is the new node joining as the predecessor of node d. The node d splits its user address space and transfers half over to c. Note that it actually transfers all the records in its address space as c will also be a replica of d. Similarly, node b also transfers the records in its address space for c to replicate. Finally, node d notifies e to stop mirroring the portion of d's space that has been transfered to c.

The general recovery mechanism for $k \geq 1$ is similar to the above outlined procedure. If we assume that c is the position of the new node which joins as the predecessor of the node whose address space is split, then using the two operators previously, we can write the general algorithm for node joins as:

```
for i = 1 to k
do
  Notify s(c,i) to drop p(s(c,i),k+1) from its replica list
  Notify p(c,i) to drop s(p(c,i),k+1) from its replica list
done
Notify s(c,k+1) to drop c from its replica list
```

For the case in which c joins as the successor of the node whose address space is split, the operator in last line is changed to $p(.,.)$. During the split, the original node continues to answer queries for the entire address range until all the data has been copied over to the new node and the new node's arrival has been propagated to all nodes.

The above discussion is based on the overlay already existing. This raises the question of how to to initialize or bootstrap the overlay. We do this in one of two ways: either a small subset of nodes can be preconfigured with the identiites of the other nodes in the overlay and the address space partitioned among these node, or the overlay can start by consisting of only a single node. More nodes can then be added using the join procedure until the required number of nodes are in the overlay.

The second technique may result in unequal distribution of user-ids but that can be resolved as shown in the next section.

6.3.5 Dynamic Load Balancing

The LUNA service uses user-ids as the keys to store the user records on the nodes. The distribution of these user-ids can be highly non-uniform. Since the address space of the user-ids is divided among the nodes of the overlay, it is possible that certain nodes maintain a larger set of users than other nodes and may become overloaded as nodes join and leave the service. The load on a node also depends on the popularity of the user-ids on it.

To handle these user dynamics and accomodate for node joins and failures, we use a dynamic load balancing scheme in LUNA. A heavily-loaded node can reduce its load by moving a portion of its address space to either its predecessor or successor. Since the data is replicated at both nodes, this just requires a change in the address space that each of these nodes advertises and so this process is much faster and less disruptive. This process can be repeated as often as required till all nodes have approximately the same load.

6.4 Multi-attribute Queries

Until now, we described the functioning of the primary overlay that stores user records indexed by their user-ids. LUNA is designed to support querying on multiple attributes. The additional attributes that can potentially be incorporated into LUNA are a user/service name, IP address, and network and geographic location. The choice of most of these attributes is straightforward with the IP address used for providing IP address to user-id mapping and the string field being used as a "real name" or description field. We envision this field as being used by an entity, such as a company, to describe some service that is being offered for the use of VoIP users on this

overlay. These additional attributes are called secondary attributes and are not necessarily unique. We forsee that the queries on the secondary attributes such as the string field and network and geographic location attributes will mostly be range-based while the queries on the primary attribute and the IP address are likely to be specific lookup requests.

To support the secondary attributes, the nodes constituting LUNA are arranged into multiple overlays, one overlay for each attribute. All nodes are part of the primary overlay but each node also participates in one of the other attribute overlays which is chosen at random when the node joins LUNA. The secondary attribute overlays are also organized in the same manner as the primary overlay. Each node maintains network pointers to all attribute overlays. Thus, the attribute space is divided into regions and each node in the secondary overlay belongs to the region in which the address space which it stores lies. The secondary overlays use the primary overlay's region hierarchy to distribute the changes in the secondary overlays. This reduces the amount of state that is required for the secondary overlays. When a new node joins or a node fails, the recovery mechanisms function exactly as in the primary overlay. Essentially, each overlay runs its own recovery and load balancing procedures that behave in the same manner.

6.4.1 Query Routing with Multiple Attributes

Each node in LUNA can accept queries for records matching any of the attributes stored in LUNA. When a query for a particular attribute arrives at a node, it routes it to the appropriate attribute overlay where the query is processed in a manner similar to the primary overlay. For range-based queries, the query is routed by the proxy node to all the nodes that lie in the range of the query. When the proxy receives all the responses, they are aggregated and the aggregated response is returned to the user.

When a new record is to be inserted into LUNA, the node inserting the record make a query in each of the attribute overlays for the particular record. The query is routed to a set of nodes, one in each attribute overlay. The record is inserted into each of these nodes, either by copying the record or by storing it at one of the nodes (the home node of the user-id) and storing pointers to the record at the other nodes.

6.5 Evaluation

6.5.1 Methodology

All our simulations are performed using a modified version of the *p2psim*, a peer-to-peer simulator [29]. We use this simulator to provide an event-based message level simulation of the LUNA service. We use two different publicly available network topologies for our experiments. The first topology is a dataset of inter-server measurements of 1740 DNS servers obtained using King [31] and is provided with the p2psim simulator. The second topology is composed of 2500 nodes used in Meridian [84]. To analyze the performance of LUNA on networks of different sizes, we use the King dataset to derive two smaller topologies of sizes 200 and 950 nodes by randomly select a set of nodes and using the King dataset to obtain the inter-server latencies of the nodes composing the set. These topologies provide realistic inter-node latencies for the nodes that form the LUNA network. This allows us to obtain results that represent a network distributed across the Internet similar to that of the DNS system. Since the Meridian topology is from a different source, its characteristics such as average RTT between the nodes is different (75ms as opposed to 174ms) from the topologies derived from the King dataset. Thus the latency measurements when using the 2500 node topology are not directly comparable to those of the other topologies.

In our experiments, we investigate the scalability of LUNA to millions of users over different network sizes. The primary metrics that we use for the evaluation of LUNA are the response times of the queries and updates within LUNA and the bandwidth

used by the nodes to provide this service. The response time that we report on in our experiments is the time taken for a query or update generated by a proxy node to get a response back as we are primarily interested in the delays imposed by the LUNA architecture. We assume that the LUNA nodes are widely dipersed and most clients are "close" to a LUNA node in terms of latency. For this reason, we do not explicitly model the latency between a client and its proxy node.

LUNA is designed to run on nodes that are relatively robust, in that node failures are rare. For this reason, in our experiments we have a one node failure event and a single node join event. In all our experiments, we begin by assigning user records to the nodes in the overlay. The number of user records assigned to each node is drawn from a normal distribution with mean set to the average number of user per node for the network. We also assign the neighbours and the replicas for each of the nodes at this time. The simulation is then allowed to run and stabilize for 300 seconds, at which time one node is failed and the system is allowed to recover from the failure. At 875 seconds a new node joins the network and the simulation ends at 1500 seconds. Unless specified otherwise, all of the following experiments were run using the 950 node topology with 2 replicas and 3 caches.

Each node in LUNA can act as the proxy node for the users connecting to it. Thus, during the course of the simulation, each node generates queries for user records stored on other nodes (simulating the queries from users connected to it). If a query fails because the target node is down, it is re-tried upto two more times to different nodes responsible for the record. Each node also generates updates for the user records of the users currently connected to it (simulating the updates generated by the clients). The update and query rates can vary widely depending on the application that uses the LUNA architecture. To analyze the performance of LUNA, we model the query and updates rates of two different scenarios, one with a high query rate and low update rate which models a DNS workload and the second with a low query

rate and a high update rate modelling a Third Generation Partnership Program - IP Multimedia Subsystem (3GPP IMS) network. These two scenarios potentially represent the extremes of workload that LUNA will be used for and allow us to examine its behaviour under maximum load.

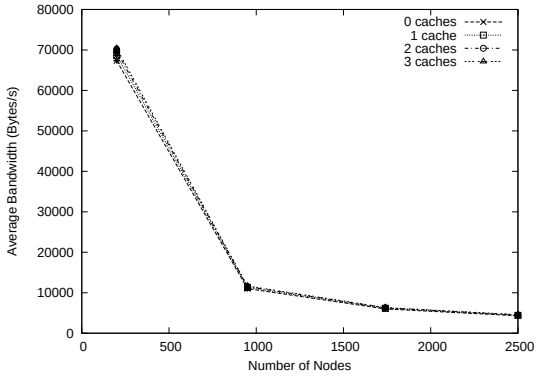
For each graph in our experiment, we plot the average value obtained over three runs of the simulator with different seeds. We also plot the minimum and maximum values obtained as the error bars on the graphs. When these bars are very close together, it indicates that the three runs produced very similar values, while larger bars indicate wider variation in the reported values.

6.5.2 DNS Workload

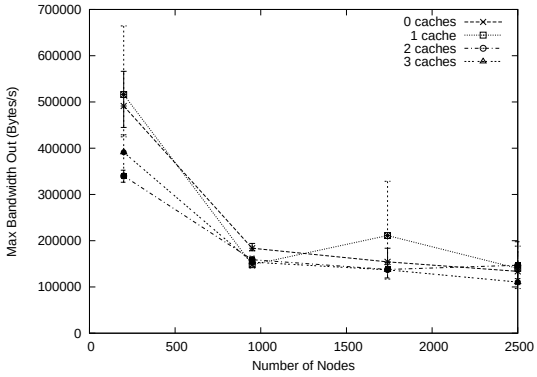
We begin by describing the results for the DNS workload. This workload is designed to model a system with a very high query rate and low update rate similar to the DNS system. The high query rate models frequent access to data whose mapping does not change frequently. The query distribution is modeled using a Zipf distribution with an α value of 0.91 [37]. Based on the experimental setup, the inter-query time is drawn from an exponential distribution with a mean as low as 10ms at each node.

6.5.2.1 Scalability

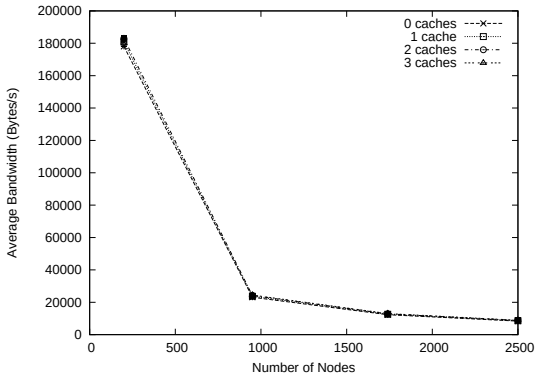
We begin by examining the performance of LUNA as we increase the number of nodes in the network. In Figures 32(a) and 32(c), we plot the average bandwidth required per node when the network handles 10 and 20 million users respectively. It is clear that as we increase the number of nodes the average bandwidth required per node decreases. The decrease in required bandwidth going from 200 to 950 nodes is significant but as the number of nodes in the network increases beyond 950, the reduction in required bandwidth is much less. This large reduction can be explained by the number of users handled by each node in the LUNA network. The average number of user records per node for the 200 node network is 50000 while



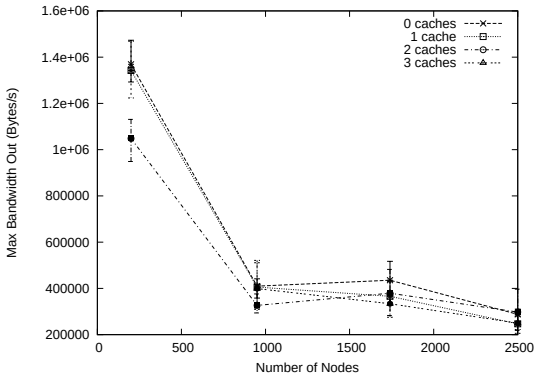
(a) Average Bandwidth per node for 10 million users



(b) Maximum Bandwidth per node for 10 million users



(c) Average Bandwidth per node for 20 million users



(d) Maximum Bandwidth per node for 20 million users

Figure 32: Scalability of LUNA with number of nodes

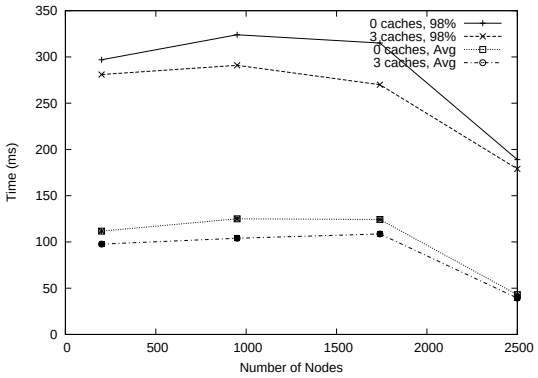
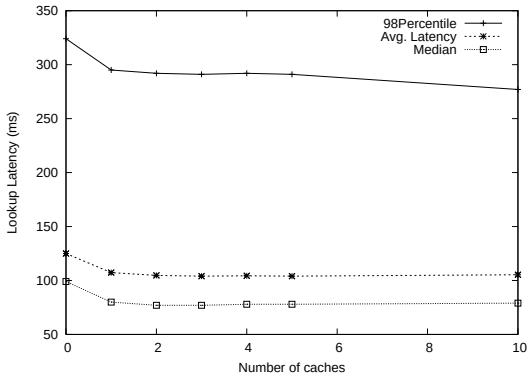


Figure 33: Query response latency for 10 million users

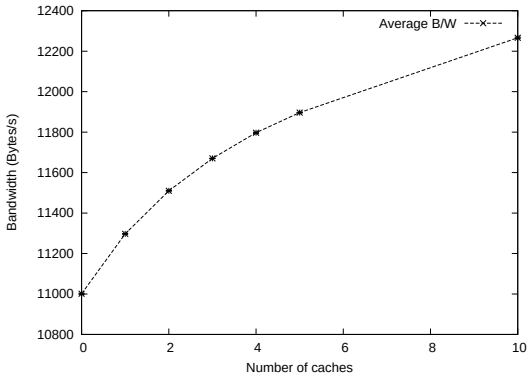
that for the 950 node network is approximately 10500. Thus, the five-fold reduction in the number of users results in an approximately six-fold reduction in the average bandwidth required. Thus, the average bandwidth required scales somewhat linearly with the number of user records stored on a node. Moreover, the bandwidth required is also consistent across the different runs with the difference between the runs being almost negligible. We also observe that the average bandwidth doesn't appear to change significantly as we increase the number of caches but that is primarily due to the scale of the y-axis. We investigate the effects of caching in more detail in Section 6.5.2.2.

In Figures 32(b) and 32(d), we plot the maximum bandwidth observed during a run for 10 and 20 million users respectively. The maximum bandwidth required is extremely variable but shows a decreasing trend as the number of nodes in the network increases. The maximum bandwidth is affected by the pattern of queries for the users and the overhead of dealing with node failures and joins. We investigate these causes in Sections 6.5.2.5 and 6.5.2.3.

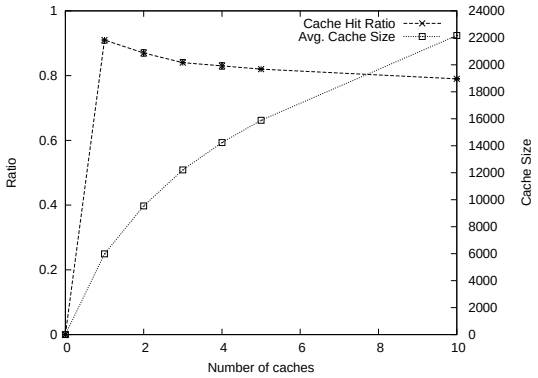
In Figure 33 we plot the average and 98-percentile query response latency for 10 million users as we increase the number of nodes. We observe that the response time remains relatively unchanged except for the 2500 node network. Note that this network was obtained from a different dataset with a different inter-node latency



(a) Latency of query response



(b) Average Bandwidth per node



(c) Cache Hit Ratio and Average Cache size

Figure 34: Varying the number of caches for a 950 node LUNA network with 10 million users

(75ms as opposed to 174ms). Note that the average latency is less than the average inter-node RTT of the networks, which indicates that the selection of the closest cache or replica is effective. The 98-percentile values are approximately three times the average values due to the way the query handles node failures. Whenever the first query fails, a node retries the query to two other nodes, leading to the factor of three.

Based on our observations of the linear behaviour of the network, we will use the 950 node network with 10 million users as the representative network for our subsequent experiments.

6.5.2.2 Number of caches and replicas

To investigate the effect of using caches, we run experiments with the number of caches varying from 0 to 10. The query response latency is plotted on the y-axis in Figure 34(a) while the average required bandwidth is plotted in Figure 34(b). The effect of having more caches is seen in the reduction in the average and median query response times. We observe that there is a significant reduction when going from no caches to 1 cache, but beyond three caches there is little change in the average or median response time. The 98 percentile value continues to decrease as we increase the number of caches. This is primarily due to the increase in the locations at which the user record is stored, with more locations increasing the probability of finding a "close" node. Note that this is also dependent on the cache selection policy. It is possible to *increase* the latency using caches using a poor selection of caches. We observe that the required bandwidth increases with the number of caches. This is a consequence of the overhead of updates sent from the home node to the caches when the user record is changed as well as the query response traffic generated by the nodes.

We also plot the cache hit ratio and the average cache size at each node in Figure 34(c). The hit ratio is plotted on the left y-axis and decreases as we increase the number of caches. This is primarily a consequnce of the way caching is currently implemented in LUNA and the nature of the query access pattern. Since the queries are generated using a Zipf distribution, many of the user records in the tail have only few queries. Since the caches are empty when the simulation starts, the initial query to a cache always results in a cache miss. As the number of caches increase, this causes the number of misses to also increase. The average cache size, which is the total number of user records cached at a node, also increases with the increase in the number of caches. As the number of caches for a node increases, each node acts as a cache for increasing number of nodes, which in turn, leads to the increase in the

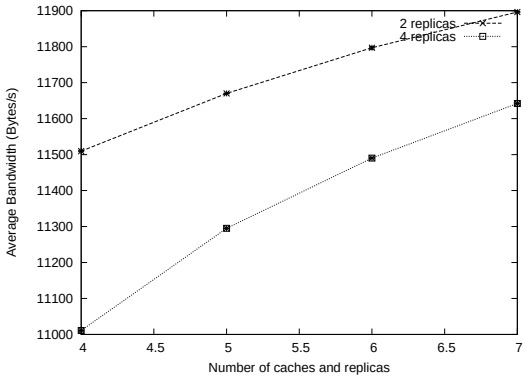


Figure 35: Varying the number of replicas

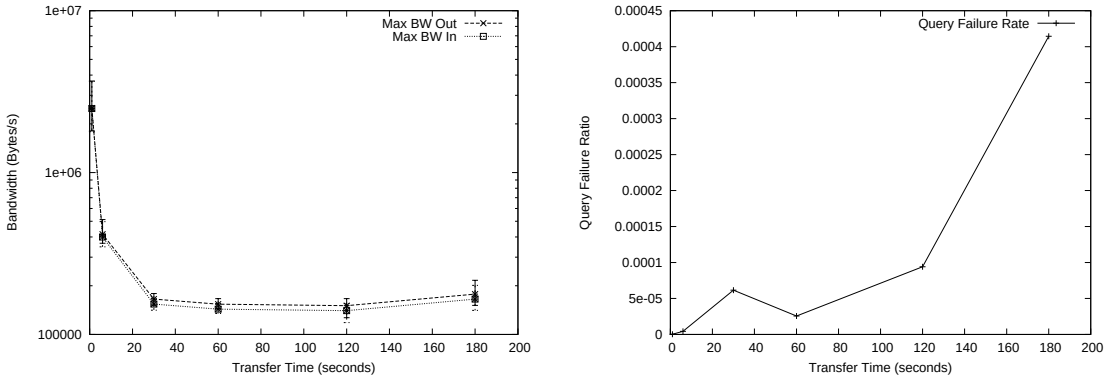
average cache size.

To gauge the effect of increasing the number of replicas, we run experiments in which each node maintains two predecessors and two successors for a total of four replicas. Note that in this case, two additional nodes now have the same user record information. To make this scenario comparable to previous results, in which the number of predecessors and successors was limited to one each, we compare based on the total number of locations at which the user information is available, i.e., the sum of the caches and the replicas. In Figure 35, we plot the average bandwidth required on the y-axis as we increase the total number of replicas and caches on the x-axis. We observe that for this workload, increasing the replicas from two to four reduces the average bandwidth by a small amount.

6.5.2.3 Effect of Transfer Time

The transfer time is used to determine the time taken to transfer user records during failure recovery and node join operations. To determine the effect of this variable, we vary the transfer time from one second to 3 minutes. Note that the actual join or recovery operation is several multiples of the transfer time with the actual time taken depending on the number of replicas in the LUNA network.

We plot the maximum bandwidth required on the y-axis as we vary the transfer time on the x-axis in Figure 36(a). We observe that with short transfer times (1-30



(a) Maximum Bandwidth (b) Query failure rate

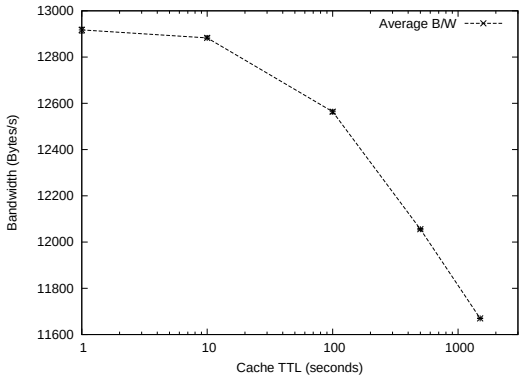
Figure 36: Effect of Transfer Time on the LUNA network

seconds), the recovery/join operations dominate the maximum bandwidth required. Beyond 60 seconds, the maximum bandwidth remains relatively constant indicating that the bandwidth required to perform the record transfers is less than the maximum bandwidth required in the normal operation of the LUNA network.

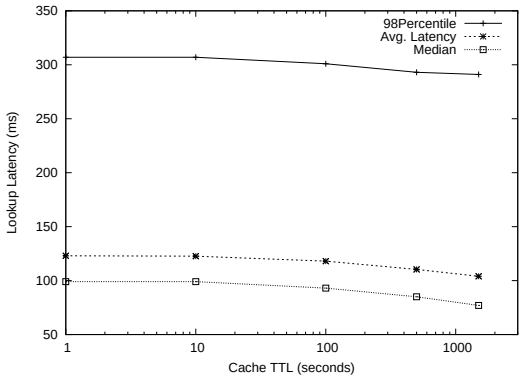
One consequence of the increase in the transfer time is that the join and recovery operations take longer to complete. This can affect the queries to the records that are hosted on the failing or joining node. For this, we plot the query failure ratio (the number of failed queries to the number of answered queries) as the transfer time increases in Figure 36(b). We see that there is an increase in the number of failed queries as expected, but the actual number of queries remains extremely small indicating that the replication and the caching mechanisms of LUNA work well.

6.5.2.4 Cache Expiration Period

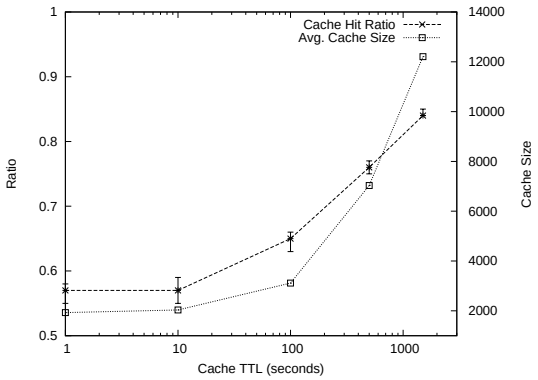
The LUNA network uses both replicas and caches to reduce the latency of the query responses and also to reduce the load on the home nodes. The replicas contain all the information stored on the home node whereas the caches store only the user records that have been requested through them. To limit the resource usage of caches, there exist a wide range of strategies to expire cache entries. To explore the effect of the cache expiration times on the performance of the LUNA network, we choose a simple



(a) Average Bandwidth



(b) Latency of query response



(c) Cache Hit Ratio and Average Cache size

Figure 37: Effect of the cache expiration period (CacheTTL)

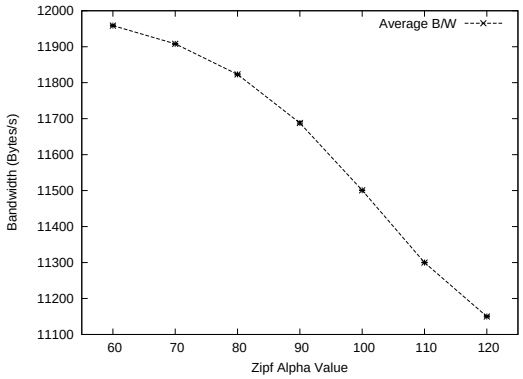
popularity metric. We count the cache hits for the entries during each period and expire those that are not used in that period, i.e., whose hit count is zero.

We plot the average bandwidth required and the query response latency in Figures 37(a) and 37(b) respectively. Note that the x-axis is plotted on log-scale. We vary the cache expiration period (CacheTTL) from one second all the way to 1,500 seconds which is the total simulation time. The very large value means that no entries are removed and allows us to estimate the total amount of resources required to maintain the cache. We observe that the average bandwidth required reduces as the CacheTTL is increased. The query response times (average, median and 98 percentile) also decrease with the increase in CacheTTL. Both these behaviours can be explained by the cache hit ratio plotted in Figure 37(c). As the CacheTTL time is increased, the cache hit ratio also increases. This implies that more queries are answered by the caches, reducing the load on the home node and therefore, the average bandwidth. The cache hit also means that the extra round trip to the home node is avoided, reducing the response time.

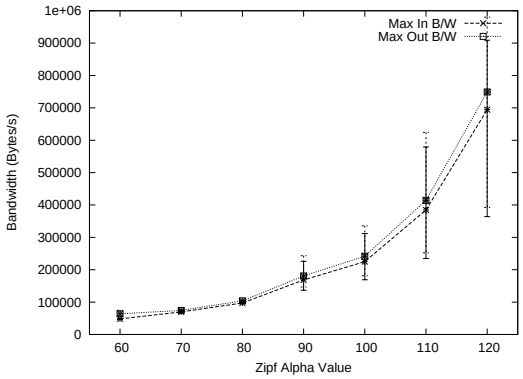
6.5.2.5 Query distributions

In all our experiments in this section, we have used a Zipf distribution to model the query distribution which is widely accepted as a reasonable model for queries on the Internet. While LUNA performs well under this query distribution, the sensitivity of LUNA's performance to the parameters of the distribution is not clear. To answer this, we run several simulations varying the α parameter of the Zipf distribution.

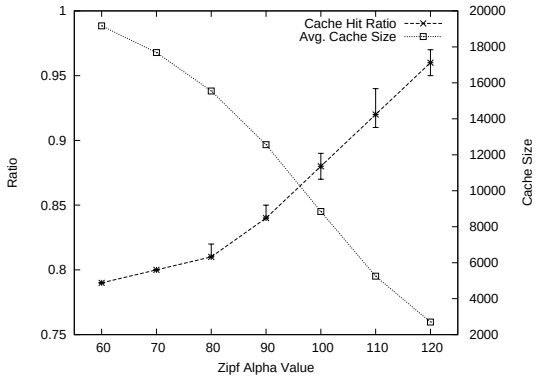
In Figure 38(a), we plot the average bandwidth required as a function of the α parameter as it varies from 0.6 to 1.2. This increase in α corresponds to the distribution decaying faster. We observe that the average bandwidth decreases as α increases, but as shown in Figure 38(b), the maximum bandwidth increases. As α increases, more queries are concentrated on the most popular user records. This



(a) Average Bandwidth



(b) Maximum Bandwidth



(c) Cache Hit Ratio and Average Cache size

Figure 38: Effect of variations in query distribution

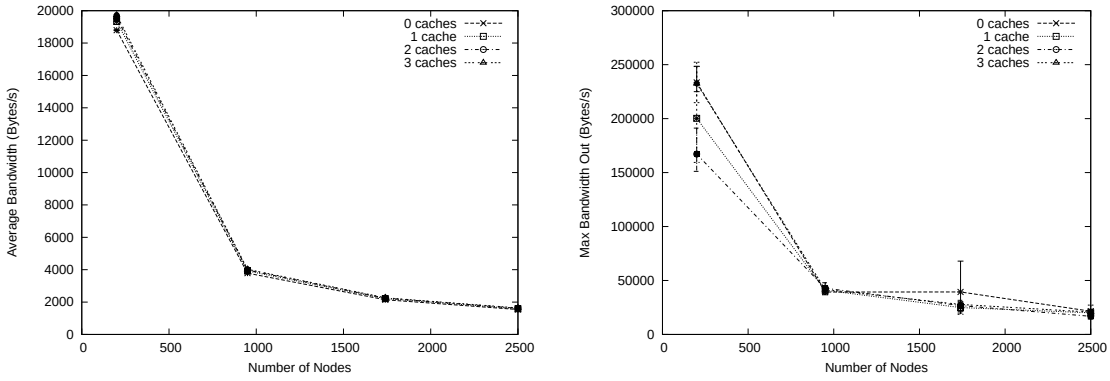
behaviour results in the nodes hosting the popular records getting a larger share of the query traffic, while reducing the query traffic to the other nodes. This results in a lower average bandwidth while increasing the maximum bandwidth at the nodes with the popular records.

In Figure 38(c), we plot the cache hit ratio as α increases and we see the effect of the popular records once again as the hit ratio increases with the increase in α. When α is low, the queries are distributed among more popular nodes, leading to more cache misses and a relatively lower cache hit ratio. When α is high, many queries are for the popular nodes which are already cached, leading to higher hit ratios. This is also reflected in the cache sizes, since the higher α leads to fewer popular user records which need to be cached.

The results from this section argue for a more aggressive caching strategy for the popular user records. Spreading the load caused by the popular records would lead to a decrease in the maximum bandwidth required per node which in turn would make the bandwidth costs of running LUNA lower at the expense of a slight increase in the average bandwidth. We leave the investigation of this as future work.

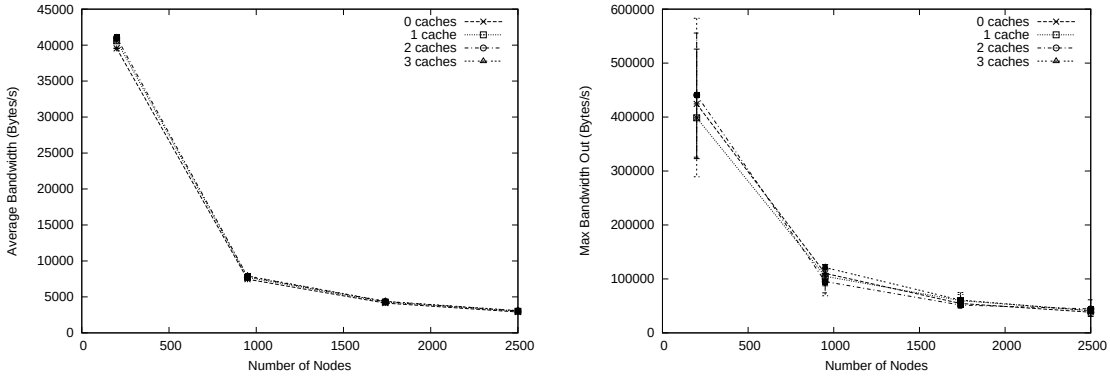
6.5.3 IMS Workload

The IMS workload is characterized by a high update rate and low query rate. The high update rate of user records models changes to the user's network location when the user is connected to the IMS network. For the IMS workload the query rate is lower and models a phone call setup in which a lookup for the destination user is made before the call is setup. Based on the IMS standard [1], we chose the Zipf distribution with an α of 0.82 to model the query distribution. The standard specifies that the average query rate is 3 per user per hour. We translate that rate into an inter-query rate for our simulator. This inter-query time is drawn from an exponential distribution with a mean of 12ms for each node in a 200 node topology and 57ms in



(a) Average Bandwidth per node for 10 million users

(b) Maximum Bandwidth per node for 10 million users



(c) Average Bandwidth per node for 20 million users

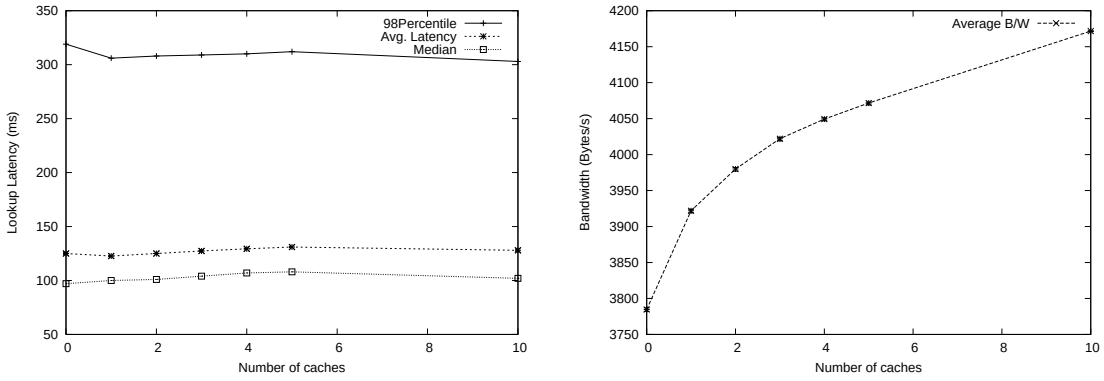
(d) Maximum Bandwidth per node for 20 million users

Figure 39: Scalability of LUNA with number of nodes on IMS workload

a 950 node topology. The updates are distributed uniformly among the user records. The IMS standard specifies a rate of 6 updates per user per hour corresponding to 60ms between updates at each node. We have performed experiments with rates of 12 and 24 updates per user per hour (corresponding to 30ms and 15ms) as well. Since the performance of LUNA scales linearly, we have omitted these results.

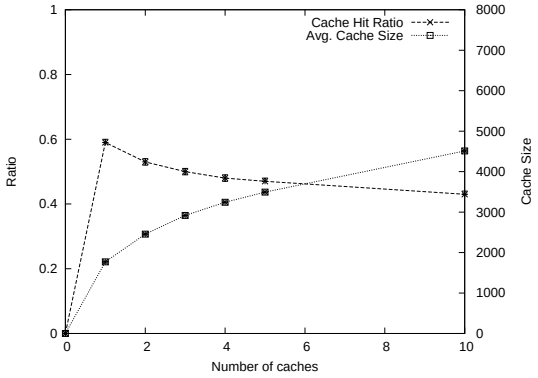
6.5.3.1 Scalability

We plot the average bandwidth required per node when the network handles 10 and 20 million users respectively using the IMS workload in Figures 39(a) and 39(c). The results of these experiments are very similar to the DNS workload. The important difference is that the average bandwidth required in this case is much lower than that



(a) Latency of query response

(b) Average Bandwidth per node



(c) Cache Hit Ratio and Average Cache size

Figure 40: Varying the number of caches for a 950 node LUNA network with 10 million users

of the DNS workload. This is primarily a function of the reduced query rate of the IMS workload. The queries in the IMS workload reduce by a factor of 15 from the DNS workload, from approx 150 million queries to 10 million. On the other hand the updates increase from approximately 200,000 to 30 million for the IMS workload.

Similar conclusions can be drawn from the plots for the maximum bandwidth in Figures 39(b) and 39(d). The majority of the bandwidth in the DNS workload is used by the query traffic whereas the IMS workload is composed more of user record update traffic.

6.5.3.2 Number of caches and replicas

The effect of increasing the number of caches is similar to that of the DNS workload except that due to the reduction in the number of queries, the query response latency

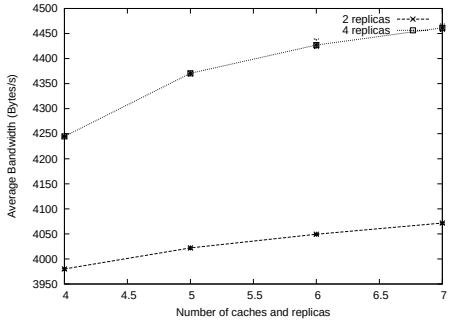
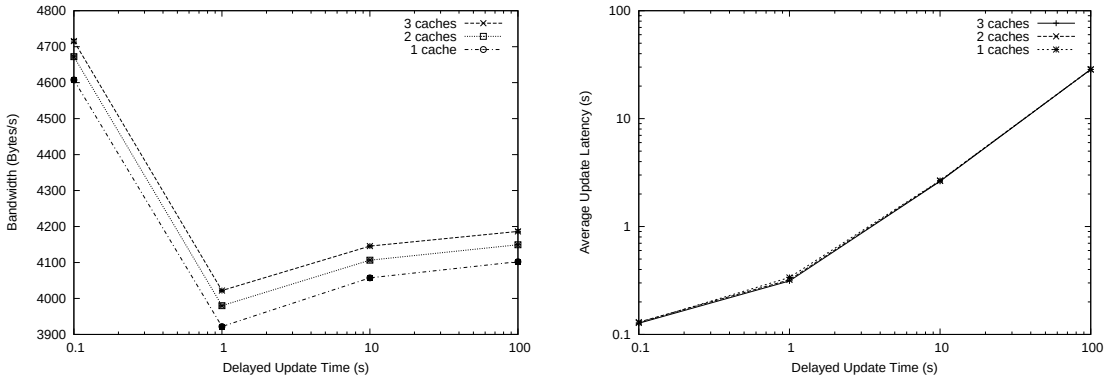


Figure 41: Varying the number of replicas

in Figure 40(a) remains relatively unchanged as the number of caches is increasd. The cache hit ratio in Figure 40(c) is also lower than that of the DNS workload. This is due to two factors, namely the value of α in the Zipf distribution chosen for the IMS workload and the reduction in the number of queries.

From the results of Section 6.5.2.5, we know that a value of α of 0.82 would result in a cache hit ratio of approximately 0.8 with a DNS workload. The number of cache misses for both distributions would be similar since the caches are empty when the simulation starts. Given that the number of queries in the IMS workload is much less, the number of hits cannot increase as in the DNS experiments. This results in a lower hit ratio as shown in the graph.

In Figure 41 we explore the effect of increasing the number of replicas of each node. We observe that using two replicas is better than four as opposed to Figure 35. The large increase in the updates for the IMS workload increases the bandwidth required to keep the replicas in sync. Caches do not suffer this problem as only updates to cached entries have to be propagated. This can be observed in the magnitude of the increase in Figure 40(b) where the increase in bandwidth is less than 300 bytes/s for 10 caches as compared to more than 250 bytes/s when we convert two caches to replicas in Figure 41.



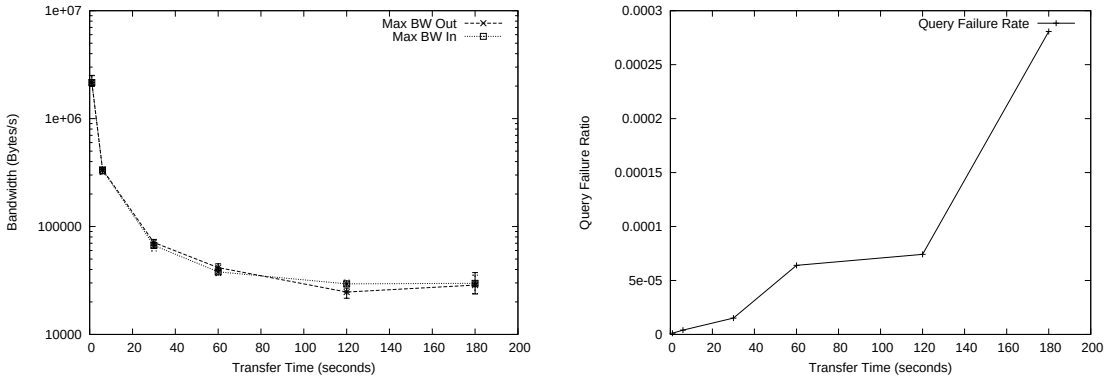
(a) Average Bandwidth (b) Update Latency

Figure 42: Overhead of updating caches and replicas on the LUNA network

6.5.3.3 Overhead of updating caches and replicas

Whenever a user record stored at the home node is updated by the user, this change has to be propagated to the all replicas of the home node and to all caches that are currently storing this record. These update messages have an overhead of protocol packet headers for each packet sent out by the home node. To reduce this overhead, we implemented a simple mechanism, similar to delayed ACKs in TCP, in which the updates are held for a short time in case other updates arrive which can be sent together in a single packet.

In Figure 42(a), we plot the average bandwidth as we vary the time the updates are delayed (note that the x-axis is on log scale). We plot the average update latency (time for the update to reach the caches and replicas) in Figure 42(b). There is a significant overhead in sending out the updates as soon as they arrive as the 0.1 second value has the largest average bandwidth requirement. On the other hand, delaying the updates causes the update to reach its destination later and increases the window in which the records are out of sync. Delaying the updates by upto one second gives a reasonable balance between the bandwidth required and update delays.



(a) Maximum Bandwidth (b) Query failure rate

Figure 43: Effect of Transfer Time on the LUNA network

6.5.3.4 Effect of Transfer Time

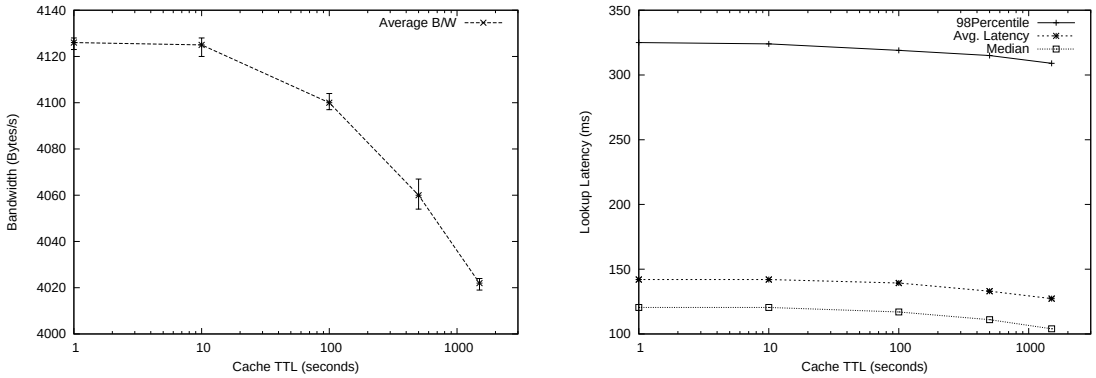
The variation in transfer time is expected to have a larger effect on the IMS workload than the DNS due to the much larger number of updates. From Figure 43(a), we observe that the maximum bandwidth follows a similar trend to that of the DNS workload, but since the query traffic is reduced drastically, the maximum bandwidth required is almost a factor of 4 less than the DNS workload. Figure 43(b) shows that the ratio of failed to successful queries is less than that of the DNS but remains similar in trend.

6.5.3.5 Cache Expiration Period

The cache expiration period plays a similar role as in the DNS workload with the average bandwidth (Figure 44(a)) and query response latency (Figure 44(b)) both reducing with increasing cache expiration periods. The cache sizes and hit ratios are lower than that of the DNS workload but have a similar trend.

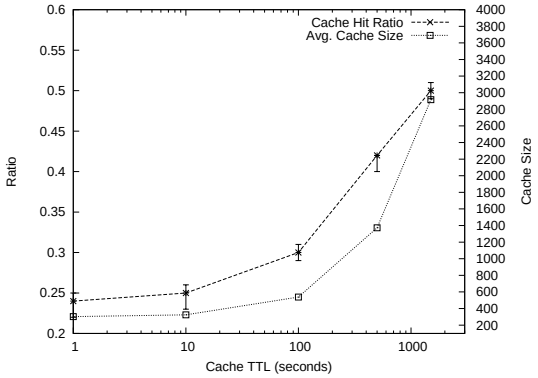
6.5.4 Comparison to other approaches

LUNA provides a distributed resource location service. This type of service is typically offered by creating large clusters of machines in several locations, each with a complete copy of the data. This cluster approach usually requires some specialized hardware



(a) Average Bandwidth

(b) Latency of query response



(c) Cache Hit Ratio and Average Cache size

Figure 44: Effect of the cache expiration period (CacheTTL)

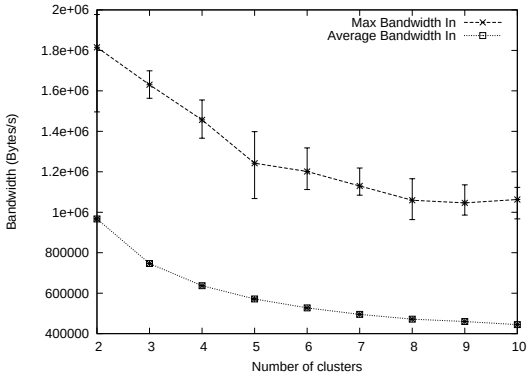


Figure 45: Varying the number of clusters for 10 million users

that is fault tolerant or a mechanism to switch between nodes in a cluster if any node fails. Scaling this approach requires investing in more machines per cluster or increasing the number of clusters, with the latter being significantly more expensive.

In Figure 45, we plot the average and maximum bandwidths required against the number of clusters used for serving the records of 10 million users. Comparing the maximum and average bandwidths required for LUNA in Figures 39(b) and 39(a) against this figure shows a factor of 7 increase in the maximum bandwidth required for the cluster approach and almost a factor of 20 for the average bandwidth.

Clearly, LUNA uses significantly less bandwidth per node, but more importantly can be implemented using off-the-shelf hardware and can scale easily as the number of users increases. This use of low cost hardware is the single biggest advantage of LUNA over the traditional cluster approach.

6.6 Summary

We have described Luna, a distributed naming service based on a one-hop overlay. Luna uses limited replication to support high availability, striving for 99.999% using inexpensive computer systems. We have shown that sporadic crashes and joins cause negligible impact on the overall query and update latency. LUNA is also designed to process very frequent updates of the records stored in it. Furthermore, LUNA is designed to answer certain sets of range queries, and can accomodate multi-attribute queries using multiple distributed indices.

In our simulations using the p2psim simulator, we have quantified the bandwidth requirement for LUNA network. Since the nodes are mostly stable, a modest amount of bandwidth is required to update all nodes during the maintenance of the global routing table used by the one-hop overlay. We have explored the effects of using replicas and caches and shown that a combination of them gives the best performance in terms of bandwidth and query latency.

CHAPTER VII

CONTRIBUTIONS

Overlay networks are the prefered way to deploy new services and improve existing services on the Internet. Managed overlays are a class of overlay networks consisting of networks in which the nodes, their capabilities and locations are known to each of the nodes in the network. Managed overlays offer the possibility of node placement and network design due to the global knowledge available. In this thesis, we have proposed two algorithms for design of managed overlay networks.

In our first algorithm, we proposed a method to optimize access to the uplink bandwidth of overlay nodes participating in an application-layer multicast tree. We showed that our method, called Time Division Streaming, can be used to schedule access to the link iamong competing overlay links to reduce the average time to transfer real-time content. TDS works by allowing each overlay link, in turn, to sent large blocks of data through the access link.

Our second algorithm, RouteSeer applies to service overlays composed of dedicated nodes. RouteSeer is used to place intermediate nodes to protect the links between overlay nodes by providing alternate paths through the intermediate nodes. These disjoint alternate paths can also be used to improve the performance of the overlay.

By extending the network model that we developed for RouteSeer, we examine the general problem of path diversity between pairs of nodes. We examine and compare other approaches to path diversity, including multihoming and proposals for multi-path routing using BGP. We showed that overlay networks offer the greatest flexibility in providing path diversity. We evaluated the performance of managed overlays using network characteristics such as packet loss and latency. We showed

that managed overlays can recover from packet losses without being penalized on latency performance.

Using some of the ideas outlined above, we designed a managed overlay to provide a resource location service that maps identifiers to network locations. The service, called LUNA, was designed to be extremely scalable and have the ability to handle highly dynamic user data. We evaluated our design and showed its advantages over other approaches.

7.1 Future Directions

This thesis focuses on techniques for designing managed overlay networks. The techniques in this thesis can be extended in two different directions. First, RouteSeer can be used on existing networks to provide a measure of overlay link resiliency. This metric of resiliency can be used as a basis for evaluating current overlay design techniques and identify links in these networks that need to be redesigned to improve resiliency.

Another direction to extend work from this thesis is to examine the interaction of overlays with the underlying network. Managed overlay networks are currently designed without consideration to the underlying BGP policies imposed by the network operators to manage the traffic in and through their networks. For example, AS policies can be configured to allow transit for traffic originating from a set of client ASes while denying transit to other ASes. The second direction is to extend the basic RouteSeer technique outlined in this thesis to design managed overlay networks that respect the various traffic policies instituted by network operators in their ASes. One approach to designing such networks is to use RouteSeer to create a pool of potential locations from which nodes that satisfy the AS policies can be selected as the actual overlay nodes.

APPENDIX A

PROOFS OF TIME DIVISION STREAMING LEMMAS

We first outline some notation for the subtrees of nodes that will be used in the following proofs. We use S_x^i to denote the x^{th} subtree of node i and $N_{S_x^i}$ to denote the number of nodes in the subtree. As in the previous sections, we use $c(i)$ to denote the number of children of node i and $d(i)$ to denote the maximum degree bound of i.

Proof of Lemma 2. Assume, for the sake of contradiction, that the optimal TDS tree has a node p which has adjacent child nodes i and j, such that node i is sent data before node j but i's subtree is smaller than j's subtree. Consider the change in finish times if we instead send data to node j before i. The finish times of the node i and its entire subtree will be increased by a factor of ns/B_p while the finish times of node j and its entire subtree are reduced by ns/B_p. Since the number of nodes whose finish times are reduced is greater than the number of nodes whose finish times are increased, the overall average finish time of the entire tree is reduced. □

Proof of Lemma 3. Assume, for the sake of contradiction, that the TDS multicast tree in Figure 46 is an optimal tree with nodes i and j, such that $d(i) < d(j)$ but i is sent data before j. We show that a tree with a smaller average finish time can be constructed in which node j is sent data before node i. Without loss of generality, we assume that the time at the parent node r before sending the block of data ns to node i is zero. From our analysis of TDS, we know that the finish time of a non-root node is dependent on the finish time of its parent. Therefore, if the finish time of the root of a subtree changes, the finish time of all other nodes in the subtree change correspondingly, and hence the average finish times. In the following construction,

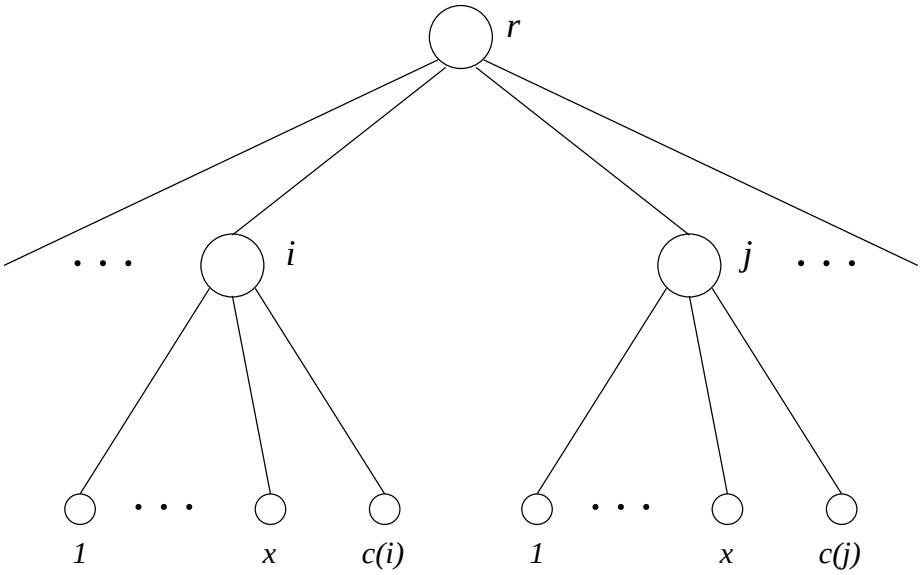


Figure 46: TDS tree before swapping positions of nodes i and j

we exchange complete subtrees between the affected nodes which results in the finish times of the root nodes of the subtrees changing.

The new tree is constructed in the following manner. The nodes i and j are exchanged and the subtrees of nodes i and j are assigned as follows:

- If the number of children of j, $c(j)$ is greater than that of i, i.e., $c(j) > c(i)$, the $c(j)-c(i)$ subtrees remain attached to node j at their original positions. Clearly, the finish times of the nodes in these subtrees are improved by ns/B_r from the original tree and so the average finish time is also lower for these subtrees.

- If $c(j) < c(i)$, then the $c(i) - c(j)$ subtrees are moved from node i to node j. Note that this is always possible as $d(j) > d(i)$. In this case, the finish times of the nodes in these subtrees changes by a factor of $nsl(B_j - B_i)/B_iB_j$ from nsl/B_i to nsl/B_j where $l \in [(c(i) - c(j) + 1, \ldots, c(i)]$. Since $d(j) > d(i)$ implies that $B_j > B_i$, the finish times and hence the average finish times of these subtrees is earlier.

For $x \in [1, min(c(i), c(j))]$, the subtrees S_x^i and S_x^j are assigned to the nodes i and

j based on their respective sizes.

If $N_{S_x^j} > N_{S_x^i}$, S_x^j is assigned to node j and S_x^i is assigned to node i. The finish time of subtree S_x^j in the original tree is

$$2ns/B_p + s/B_j + (x-1)ns/B_j$$

while the finish time in the new tree is

$$ns/B_p + s/B_j + (x-1)ns/B_j.$$

The change from the original to the new tree is ns/B_p. Similarly, the finish time of S_x^i in the original tree is

$$ns/B_p + s/B_i + (x-1)ns/B_i$$

and in new tree is

$$2ns/B_p + s/B_i + (x-1)ns/B_i.$$

The change in this case is $-ns/B_p$ but since the size of this subtree is smaller, the net gain in finish times is positive.

For the case in which $N_{S_x^j} < N_{S_x^i}$, S_x^j is assigned to node i and S_x^i is assigned to node j. The finish time of subtree S_x^j in the original tree is

$$2ns/B_p + s/B_j + (x-1)ns/B_j$$

while in the new tree, it is

$$2ns/B_p + s/B_i + (x-1)ns/B_i.$$

The change from the original to the new tree is thus

$$s(B_i - B_j)/B_iB_j + (x-1)ns(B_i - B_j)/B_iB_j.$$

For the subtree S_x^i, the original finish time is

$$ns/B_p + s/B_i + (x-1)ns/B_i$$

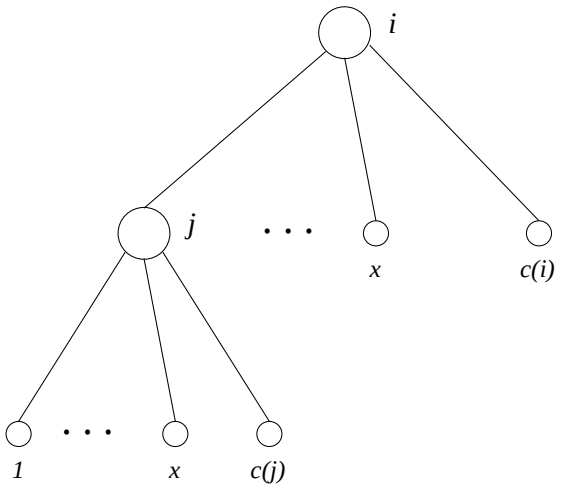


Figure 47: TDS tree before swapping positions of nodes i and j

and the new finish time is

$$ns/B_p + s/B_j + (x-1)ns/B_j$$

leading to a change in finish time of

$$s(B_j - B_i)/B_iB_j + (x-1)ns(B_j - B_i)/B_iB_j.$$

Again, due to $N_{S_x^j} < N_{S_x^i}$, the gain in the finish time of the nodes in S_x^i outweighs the increase in finish time of the nodes in S_x^j. If the subtrees are of the same size, the change in finish times of the subtrees equalize and the net change is zero.

Thus, in all cases, the new tree has a lesser average finish time than the original tree showing that the original tree could not be an optimal tree. □

Proof of Lemma 4. Assume, for sake of contradiction, that the node with the maximum degree constraint is not the root of the optimal TDS tree. By Lemma 3, this node is the first in order to receive data from its immediate parent. Consider the section of this optimal TDS tree shown in Figure 47 where $d(j) > d(i)$. We show by construction that exchanging the positions of nodes i and j in the tree will result in a TDS tree with lower average finish time. This construction can be applied as many

times as required to move the node j to the root of the TDS tree, establishing the contradiction.

In Figure 47, to show that exchanging the positions of nodes i and j results in a lower average finish time for the tree, we proceed in a manner similar to the proof for Lemma 3. WLOG, we assume that the time at the root of the given tree just before the first block is sent is zero. The nodes i and j are exchanged and the subtrees are assigned in the following manner: Let $l \in [(c(i) - c(j) + 1, \ldots, c(i)]$.

- If the number of children of j, $c(j)$ is greater than that of i, i.e., $c(j) > c(i)$, the $c(j) - c(i)$ subtrees remain attached to node j at their original positions and are moved along with the node j. Clearly, the finish times of the nodes in these subtrees are changed from $ns/B_i + s/B_j + (l-1)ns/B_j$ to nsl/B_j. The improvement in the finish times is $s/B_j + ns(B_j - B_i)/B_iB_j$.

- If $c(j) < c(i)$, then the $c(i) - c(j)$ subtrees are moved from node i to node j. Note that this is always possible as $d(j) > d(i)$. In this case, the finish times of the nodes in these subtrees changes from nsl/B_i to nsl/B_j for an improvement of $nsl(B_j - B_i)/B_iB_j$.

For $x \in [1, min(c(i), c(j))]$, the subtrees S_x^i and S_x^j are assigned to the nodes i and j based on their respective sizes.

If $N_{S_x^j} > N_{S_x^i}$, S_x^j is assigned to node j and S_x^i is assigned to node i. The finish time of subtree S_x^j in the original tree is

$$ns/B_i + s/B_j + (x-1)ns/B_j$$

while the finish time in the new tree is nsx/B_j. The change in finish times is $s/B_j + ns(B_j - B_i)/B_iB_j$. Similarly, the finish time of S_x^i in the original tree is nsx/B_i and in the new tree is

$$ns/B_j + s/B_i + (x-1)ns/B_i.$$

The change in finish times is $-s/B_i + ns(B_i - B_j)/B_iB_j$. Due to the differential in the size of the subtrees, the net gain is positive.

For the case in which $N_{S_x^j} < N_{S_x^i}$, S_x^j is assigned to node i and S_x^i is assigned to node j. The finish time of subtree S_x^j in the original tree is

$$ns/B_i + s/B_j + (x-1)ns/B_j$$

and in the new tree is

$$ns/B_j + s/B_i + (x-1)ns/B_i.$$

The change in finish times is

$$-((x-2)ns + s)(B_j - B_i)/B_iB_j.$$

For the subtree S_x^i, the original finish time is nsx/B_i and the new finish time is nsx/B_j giving a change in finish time of $nsx(B_j - B_i)/B_iB_j$. Thus, the overall change in finish times is positive, showing that exchanging the positions of nodes i and j reduces the average finish time. □

REFERENCES

[1] 3rd Generation Partnership Project, "Ip multimedia call control protocol based on session initiation protocol (sip) and session description protocol (sdp) (stage 3)." http://www.3gpp.org/ftp/Specs/html-info/24229.htm, Oct 2006.

[2] "Akamai technologies." http://www.akamai.com/, June 2005.

[3] "Akamai technologies." http://www.akamai.com/en/html/services/sureroute_for_failover.html, June 2005.

[4] Akella, A., Maggs, B., Seshan, S., Shaikh, A., and Sitaraman, R., "A measurement-based analysis of multihoming," in *Proceedings of ACM SIGCOMM*, Aug 2003.

[5] Akella, A., Pang, J., Maggs, B., Seshan, S., and Shaikh, A., "A comparison of overlay routing and multihoming route control," in *Proceedings of ACM SIGCOMM*, Aug 2004.

[6] Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R., "Resilient overlay networks," in *Proceedings of 18th ACM Symposium on Operating Systems Principles*, Oct 2001.

[7] Andersen, D. G., Snoeren, A. C., and Balakrishnan, H., "Best-path vs. multi-path overlay routing," in *Proceedings of ACM SIGCOMM Internet Measurement Conference*, (Miami, FL), Oct. 2003.

[8] Banerjee, S., Bhattacharjee, B., and Kommareddy, C., "Scalable application layer multicast," in *Proceedings of ACM SIGCOMM*, Aug 2002.

[9] Banerjee, S., Kommareddy, C., Kar, K., Bhattacharjee, S., and Khuller, S., "Construction of an efficient overlay multicast infrastructure for real-time applications," in *Proceedings of IEEE INFOCOM*, April 2003.

[10] Banerjee, S., Lee, S., Bhattacharjee, B., and Srinivasan, A., "Resilient multicast using overlays," in *Proceedings of ACM Sigmetrics*, June 2003.

[11] Banerjee, S., Pias, M., and Griffin, T., "The interdomain connectivity of PlanetLab nodes," in *Passive and Active Measurements (PAM) Workshop*, Apr 2004.

[12] Barford, P., Cai, J., and Gast, J., "Cache placement methods based on client demand clustering," in *Proceedings of IEEE INFOCOM*, Jun 2002.

[13] Bharambe, A. R., Agrawal, M., and Seshan, S., "Mercury: supporting scalable multi-attribute range queries," in *Proceedings of ACM SIGCOMM*, Aug 2004.

[14] Braynard, R., Kostic, D., Rodriguez, A., Chase, J., and Vahdat, A., "Opus: an overlay peer utility service," in *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[15] Brosh, E. and Shavitt, Y., "Approximation and heuristic algorithms for minimum-delay application layer multicast trees," in *Proceedings of IEEE INFOCOM*, March 2004.

[16] Calvert, K., Zegura, E., and Bhattacharjee, S., "How to model an Internetwork," in *Proceedings of IEEE INFOCOM*, 1996.

[17] Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., and Singh, A., "Splitstream: High-bandwidth multicast in cooperative environments," in *Proceedings of 19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[18] Cha, M., Moon, S., Park, C., and Shaikh, A., "Placing relay nodes for intra-domain path diversity," in *Proceedings of IEEE INFOCOM*, Apr 2006.

[19] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., , and Shenker, S., "Making Gnutella-like P2P systems scalable," in *Proceedings of ACM SIGCOMM*, Aug 2003.

[20] Chu, Y., Rao, S. G., Seshan, S., and Zhang, H., "A case for end system multicast," in *IEEE Journal on Selected Areas in Communication (JSAC)*, Aug 2001.

[21] Cui, W., Stoica, I., and Katz, R. H., "Backup path allocation based on a correlated link failure probability model in overlay networks," in *Proceedings of IEEE International Conference on Network Protocols*, Nov 2002.

[22] Dabek, F., Cox, R., Kaashoek, F., and Morris, R., "Vivaldi: A decentralized network coordinate system," in *Proceedings of ACM SIGCOMM*, Aug 2004.

[23] Duan, Z., Zhang, Z., and Hou, T., "Service Overlay Networks: SLAs, QoS and bandwidth provisioning," in *Proceedings of IEEE International Conference on Network Protocols*, Nov 2002.

[24] Fan, J. and Ammar, M., "Dynamic topology reconfiguration of overlay networks: Structure and approximation of optimal policies," in *Proceedings of IEEE INFOCOM*, Apr 2006.

[25] Fei, T., Tao, S., Gao, L., Guerin, R., and Kurose, J., "How to select a good alternate path in large peer-to-peer systems?," in *Proceedings of IEEE INFOCOM*, Apr 2006.

[26] "Gnutella protocol specification." http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, May 2002.

[27] Gao, L., "On inferring autonomous system relationships in the Internet," in *IEEE/ACM Transaction on Networking*, Dec 2001.

[28] Garey, M. R. and Johnson, D. S., *Computers and Intractability.* W. H. Freeman, 1979.

[29] Gil, T., Kaashoek, F., Li, J., Morris, R., and Stribling, J., "p2psim: A simulator for peer-to-peer protocols." http://www.pdos.lcs.mit.edu/p2psim/, June 2005.

[30] Gkantsidis, C., Mihail, M., and Saberi, A., "Hybrid search schemes for unstructured peer-to-peer networks," in *Proceedings of IEEE INFOCOM*, Mar 2005.

[31] Gummadi, K., Saroiu, S., and Gribble, S., "King: Estimating latency between arbitrary Internet end hosts," in *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW)*, May 2002.

[32] Gummadi, K. P., Madhyastha, H. V., Gribble, S. D., Levy, H. M., and Wetherall, D., "Improving the reliability of Internet paths with one-hop source routing," in *Proceedings of Symposium on Operating Systems Design and Implementation*, Dec 2004.

[33] Gupta, A., Liskov, B., and Rodrigues, R., "Efficient routing for peer-to-peer overlays," in *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2004.

[34] Hagens, R., Hall, N., and Rose, M., "Use of the Internet as a Subnetwork for Experimentation with the OSI Network Layer." RFC 1070, 1989.

[35] Han, J., Watson, D., and Jahanian, F., "Topology aware overlay networks," in *Proceedings of IEEE INFOCOM*, Mar 2005.

[36] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., , O'Toole, J. W., and J., "Overcast: Reliable multicasting with an overlay network," in *Proceedings of Symposium on Operating Systems Design and Implementation*, Oct 2000.

[37] Jung, J., Sit, E., Balakrishnan, H., and Morris, R., "DNS performance and the effectiveness of caching," in *IEEE/ACM Transaction on Networking*, Oct 2002.

[38] "Kazaa." http://www.kazaa.com, May 2002.

[39] "K Root Server." http://k.root-servers.org, May 2006.

[40] Krishnamurthy, B. and Wang, J., "On network-aware clustering of web clients," in *Proceedings of ACM SIGCOMM*, Aug 2000.

[41] Kumar, A., Merugu, S., Xu, J., and Yu, X., "Ulysses: A robust, low-diameter, low-latency peer-to-peer network," in *Proceedings of IEEE International Conference on Network Protocols*, Nov 2003.

[42] Kumar, A., Xu, J., and Zegura, E., "Efficient and scalable query routing for unstructured peer-to-peer networks," in *Proceedings of IEEE INFOCOM*, Mar 2005.

[43] Liston, R., Srinivasan, S., and Zegura, E., "Diversity in DNS Performance Measures," in *Proceeding of the ACM Internet Measurement Workshop*, Nov 2002.

[44] Loo, B. T., Huebsch, R., Stoica, I., and Hellerstein, J., "The case for a hybrid P2P search infrastructure," in *3rd International Workshop on Peer-to-Peer Systems*, Feb 2004.

[45] Malouch, N. M., Liu, Z., Rubenstein, D., and Sahu, S., "A graph theoretic approach to bounding delay in proxy-assisted, end-system multicast," in *IEEE International Workshop on Quality of Service*, May 2002.

[46] Manku, G. S., Bawa, M., and Raghavan, P., "Symphony: Distributed hashing in a small world," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Mar 2003.

[47] Mao, Z. M., Johnson, D., Rexford, J., Wang, J., , and Katz, R., "Scalable and accurate identification of AS-level forwarding paths," in *Proceedings of IEEE INFOCOM*, Mar 2004.

[48] Mao, Z. M., Rexford, J., Wang, J., and Katz, R. H., "Towards an accurate AS-level traceroute tool," in *Proceedings of ACM SIGCOMM*, Aug 2003.

[49] Medina, A., Lakhina, A., Matta, I., and Byers, J., "BRITE: An approach to universal topology generation," in *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug 2001.

[50] Merugu, S., Srinivasan, S., and Zegura, E., "p-sim: A simulator for peer-to-peer networks," in *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Oct 2003.

[51] Merugu, S., Srinivasan, S., and Zegura, E., "Adding structure to unstructured peer-to-peer networks: the role of overlay topology," in *Journal of Parallel and Distributed Computing*, Feb 2005.

[52] Mockapetris, P., "Domain Names - Implementation and Specification." RFC 1032, 1987.

[53] Mockapetris, P. V. and Dunlap, K. J., "Development of the Domain Name System," in *Proceedings of ACM SIGCOMM*, pp. 123–133, 1988.

[54] "Napster." http://www.napster.com, Jan. 2001.

[55] "NLANR global caching hierarchy." http://www.ircache.net/, June 2005.

[56] Nakao, A., Peterson, L., and Bavier, A., "A routing underlay for overlay networks," in *Proceedings of ACM SIGCOMM*, Aug 2003.

[57] Naor, M. and Wieder, U., "Know thy neighbor's neighbor: Better routing for skip-graphs and small worlds," in *3rd International Workshop on Peer-to-Peer Systems*, Feb 2004.

[58] "PlanetLab." http://www.planet-lab.org/, May 2004.

[59] Padmanabhan, V., Wang, H., and Chou, P., "Resilient peer-to-peer streaming," in *Proceedings of IEEE International Conference on Network Protocols*, 2003.

[60] Pendarakis, D., Shi, S., Verma, D., and Waldvogel, M., "ALMI: An Application Level Multicast Infrastructure," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Mar 2001.

[61] "Routeviews project." http://www.routeviews.org, Sept. 2006.

[62] Ramasubramanian, V. and Sirer, E. G., "The design and implementation of a next generation name service for the Internet," in *Proceedings of SIGCOMM*, Aug 2004.

[63] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM*, September 2001.

[64] Rowstron, A. and Druschel, P., "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, Nov. 2001.

[65] "CAIDA skitter." http://www.caida.org/tools/measurement/skitter/, June 2005.

[66] "Skype." http://www.skype.com, July 2005.

[67] Savage, S., Collins, A., Hoffman, E., Snell, J., and Anderson, T., "The end-to-end effects of Internet path selection," in *Proceedings of the ACM SIGCOMM*, September 1999.

[68] Shi, S. and Turner, J., "Routing in overlay multicast networks," in *Proceedings of IEEE INFOCOM*, April 2002.

[69] Shi, S., Turner, J., and Waldvogel, M., "Dimensioning server access bandwidth and multicast routing in overlay network," in *Proceedings of NOSSDAV*, Jun 2001.

[70] Sidhu, D., Nair, R., and Abdallah, S., "Finding disjoint paths in networks," in *Proceedings of ACM SIGCOMM*, pp. 43–51, 1991.

[71] Singh, K. and Schulzrinne, H., "Peer-to-peer Internet telephony using SIP," in *Proceedings of NOSSDAV*, Jun 2005.

[72] Srinivasan, S. and Zegura, E., "Scheduling Uplink Bandwidth in Application-Layer Multicast Trees," in *Proceedings of Networking 2005*, May 2005.

[73] Srinivasan, S. and Zegura, E., "RouteSeer: Topological placement of nodes in service overlays," Tech. Rep. CoC GIT-CC-06-03, Georgia Tech, 2006.

[74] Sripanidkulchai, K., Ganjam, A., Maggs, B., and Zhang, H., "The feasibility of supporting large-scale live streaming applications with dynamic application end-points," in *Proceedings of ACM SIGCOMM*, Aug 2004.

[75] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S., "Internet Indirection Infrastructure," in *Proceedings of ACM SIGCOMM*, Aug 2002.

[76] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and alakrishnan, H. B., "Chord: A peer-to-peer lookup service for Internet applications," in *Proceedings of ACM SIGCOMM*, (San Diego, CA), September 2001.

[77] Subramanian, L., Stoica, I., Balakrishnan, H., and Katz, R., "OverQos: An overlay based architecture for enhancing Internet QoS," in *Proceedings of the Symposium on Networked Systems Design and Implementation*, March 2004.

[78] Subrmanian, L., Agarwal, S., Rexford, J., and H.Katz, R., "Characterizing the Internet hierarchy from multiple vantage points," in *Proceedings of IEEE INFOCOM*, Jun 2002.

[79] Tangmunarunkit, H., Govindan, R., Jamin, S., Shenker, S., and Willinger, W., "Network topology generators: Degree-based vs. structural," in *Proceedings of ACM SIGCOMM*, Aug 2002.

[80] Tangmunarunkit, H., Govindan, R., Shenker, S., and Estrin, D., "The impact of routing policy on Internet paths," in *Proceedings of IEEE INFOCOM*, April 2001.

[81] Vieira, S. and Liebeherr, J., "Topology design for service overlay networks with bandwidth guarantees," in *IEEE International Workshop on Quality of Service*, Jun 2004.

[82] Vutukury, S. and Garcia-Luna-Aceves, J., "MDVA: A distance-vector multipath routing protocol," in *Proceedings of IEEE INFOCOM*, Apr 2001.

[83] Walfish, M., Balakrishnan, H., , and Shenker, S., "Untangling the Web from DNS," in *Proceedings of the Symposium on Networked Systems Design and Implementation*, Mar 2004.

[84] Wong, B., Slivkins, A., and Sirer, E. G., "Meridian: A lightweight network location service without virtual coordinates," in *Proceedings of ACM SIGCOMM*, Aug 2005.

[85] Xu, D., Hefeeda, M., Hambrusch, S., and Bhargava, B., "On peer-to-peer media streaming," in *Proceedings of IEEE International Conference on Distributed Computing Systems*, Jul 2002.

[86] Xu, W. and Rexford, J., "MIRO: Multi-path Interdomain Routing," in *Proceedings of ACM SIGCOMM*, Sept 2006.

[87] Yang, M. and Fei, Z., "A proactive approach to reconstructing overlay multicast trees," in *Proceedings of IEEE INFOCOM*, March 2004.

[88] Yang, X. and Wetherall, D., "Source selectable path diversity via routing deflections," in *Proceedings of ACM SIGCOMM*, Sept 2006.

[89] Zhang, B., Liu, R., Massey, D., and Zhang, L., "Collecting the Internet AS-level topology," in *ACM SIGCOMM Computer Communication Review Volume 35 , Issue 1*, Jan 2005.